Task 1.1 Define the problem

What exactly are we trying to solve with sentiment recognition?

1. In sentiment recognition, we are trying to identify and understand the emotional tone expressed in text, speech, images or any other data format (images in the assignment's case). The problem it solves is determining whether the expressed sentiment is positive, negative, neutral or more detailed such as anger, disgust, fear, happy, pain or sad, as givem in the dataset we are working on in the assignment.
2. The main goal is to nullify the gap between human emotions and machine understanding, so the softwares can -
   - Interpret the sntiment of the data expressed.
   - Classify the feedbacks, reviews or conversations automatically
   - Provide insights for decision making

What are the desired inputs and outputs of the system?

1. The desired inputs are -
   - Raw dataset that contains the six folders containing the images showing that particular emotion.
   - So, the system will take as input an individual image from one of these folders.
2. The desired outputs are -
   - A predicted label that classifies the input into one of the six emotions.

Clearly describe the target classes (Happy, Sad, Fear, Pain, Anger, Disgust)

The target class in the assignment represent six distinct human emotions, each corresponding to a folder in the datset. These are the categories that the CNN model will learn to recognize and classify from the input data:

1. Happy - Represents positive emotional state, generally expressed by a smile, joyful tone or cheerful expressions.
2. Sad - Represents negative emotional state, shown through crying or low energy.
3. Fear - Reflects a state of anxity or being threatened, visible in widened eyes, tnese body language or fearful tone.
4. Pain - Represents discomfort or suffering, often shown through facial expression or strained expressions.
5. Anger - Expresses frustration or aggression, commonly visible in frowns or glaring eyes.
6. Disgust - Represents aversion or dislike, often seen in wrinkled noses, curled lips or rejecting gestures.
   For the CNN, these six classes form the output layer, where the model predicts a probability distribution across all classes and assigns the input to the class with the

highest probability.

Why might misclassification be problematic?

Misclassification in sentiment recognition can be problematic because:

- Loss of meaning - EMotions carry critical information about the customers or humans in general. Predicting anger as happy completely changes the intended meaning.
- Poor decision-making - Many applications (e.g. healthcare, customer service, security) rely on emotion detection. Misclassification could lead to wrong actions or responses.
- User dissatisfaction - If a system malfunctions, users may lose trust in the organisation or the vision of the organisation.
- Context sensitivity - Some emotions, like pain or happy, are two completely different emotions and misclassifying them could have serious consequences (e.g. in medical or safety contexts).

In [1]:
```
import random
import numpy as np
import torch
import os

def set_seed(seed=42):
    random.seed(seed)                          # Python random
    np.random.seed(seed)                       # NumPy
    torch.manual_seed(seed)                    # PyTorch CPU
    torch.cuda.manual_seed(seed)               # PyTorch GPU (single-GPU)
    torch.cuda.manual_seed_all(seed)           # PyTorch GPU (multi-GPU)

    # For deterministic behaviour (may slow training a bit)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    os.environ["PYTHONHASHSEED"] = str(seed)

set_seed(42)
```

Task 1.2 Make a plan

Note - I have downloaded the dataset locally also, but as the dataset is very huge, it would take time to upload the dataset. So, let's use kaggle API.

In [2]:
```
!pip -q install kaggle
```

In [3]:
```
from google.colab import files
files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
```

Out[3]:
```
{'kaggle.json': b'{"username":"jp1611","key":"a25b9c0828d0bb54474f3659f71b7
9ec"}'}
```

In [4]:
```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
```

In [5]:
```
!kaggle datasets download -d yousefmohamed20/sentiment-images-classifier -p /
```

```
Dataset URL: https://www.kaggle.com/datasets/yousefmohamed20/sentiment-images
-classifier
License(s): apache-2.0
Downloading sentiment-images-classifier.zip to /content/data
 89% 101M/114M [00:00<00:00, 235MB/s]
100% 114M/114M [00:00<00:00, 252MB/s]
```

In [6]:
```
# Loading the dataset
import os
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader

data_root = "/content/data/6 Emotions for image classification"

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224,
])

dataset = ImageFolder(root = data_root, transform = transform)
```

In [7]:
```
import matplotlib.pyplot as plt
import numpy as np
import torchvision

data_loader = DataLoader(dataset, batch_size = 16, shuffle = True)

images, labels = next(iter(data_loader))

grid = torchvision.utils.make_grid(images, nrow = 4, padding = 2)

# Convert from tensor (C,H,W) → (H,W,C) for plotting
npimg = grid.numpy().transpose((1, 2, 0))

# Undo normalization (so colors look right)
mean = np.array([0.485, 0.456, 0.406])
std = np.array([0.229, 0.224, 0.225])
npimg = std * npimg + mean # reverse of the normalization
npimg = np.clip(npimg, 0, 1)

plt.figure(figsize = (8, 8))
plt.imshow(npimg)
plt.axis('off')
plt.show()
```

```
print("Labels: ", [dataset.classes[label] for label in labels])
```

```
Labels:  ['happy', 'sad', 'fear', 'sad', 'happy', 'disgust', 'happy', 'disgus
t', 'sad', 'disgust', 'sad', 'disgust', 'sad', 'disgust', 'fear', 'anger']
```

In [8]:
```
# Total number of images and images per label
from collections import Counter
print(f"Total images in the dataset are {len(dataset)}")

class_counts = Counter(dataset.targets)
for idx, count in class_counts.items():
    print(f"Class {dataset.classes[idx]}: {count} images")
```

```
Total images in the dataset are 1198
Class anger: 214 images
Class disgust: 199 images
Class fear: 163 images
Class happy: 230 images
Class pain: 168 images
Class sad: 224 images
```

1. Given the total number of images in the dataset, how would you determine an appropriate train/validation/test split ratio? Justify your choice?
   Tradationally, there are three splits, 80/10/10, 70/15/15 or 60/20/20 and the choice of the split depends on the size of the dataset.

   As our dataset has 1198, it is a small dataset and if we use 80/10/10, our test and validation sets will ahev only 120 images, which are too small to properly evaluate.

   If we use 70/15/15 split, out test and validation stes will have 180 images which is much better compared to 80/10/10.

   If we select 60/20/20, out train set will have only 720 images, which are not sufficient for goof training of the model.

   Hence, the optimla choice is to use the 70/15/15 split.

2. Using quantitative analysis (e.g., histograms, percentages), evaluate whether the dataset is balanced across the six emotion classes. What evidence indicates underrepresentation, and how might this influence learning dynamics?

   From the counts of each class -

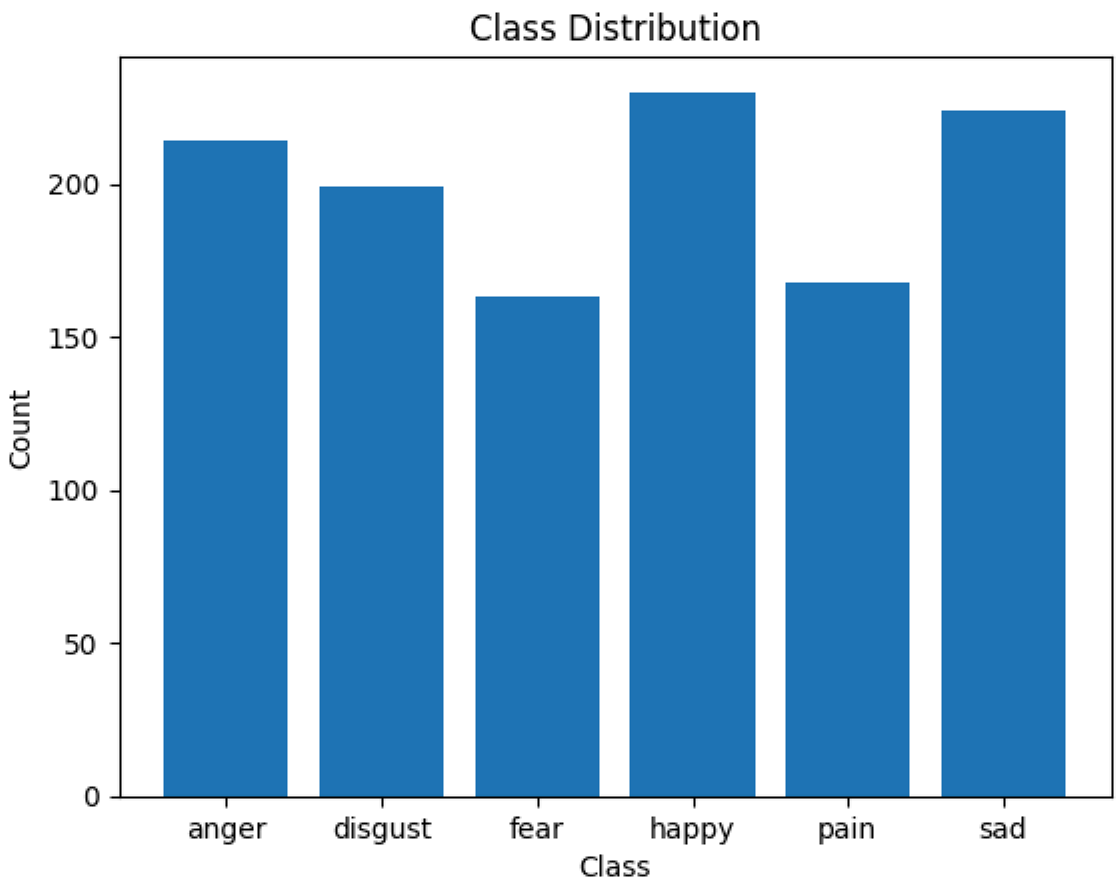
3. Class anger: 214 images
4. Class disgust: 199 images
5. Class fear: 163 images
6. Class happy: 230 images
7. Class pain: 168 images
8. Class sad: 224 images

Hence,

- anger: 214 / 1198 ≈ 17.9%
- disgust: 199 / 1198 ≈ 16.6%
- fear: 163 / 1198 ≈ 13.6%
- happy: 230 / 1198 ≈ 19.2%
- pain: 168 / 1198 ≈ 14.0%
- sad: 224 / 1198 ≈ 18.7%

The labels **fear** (13.6%) and **bold** (19.2%) are underrepresented.
happy label is overrepresented.

In [9]:
```
labels = ["anger", "disgust", "fear", "happy", "pain", "sad"]
class_counts = [214, 199, 163, 230, 168, 224]

plt.bar(labels, class_counts)
plt.xlabel("Class")
plt.ylabel("Count")
plt.title("Class Distribution")
plt.show()
```



For the underrepresented classes, the model may see fewer examples and hence, it might fail to earn their features well leading to higher misclassification rates for these classes. For overrepresented classes, the CNN may produce bias predictions twards these classes, since they appear more often.

To overcome the imbalance, we can use techniques such as **data audmentation** (oversampling, flipping, rotations, etc.) for minority classes.

3. From a theoretical standpoint, explain how class imbalance can bias gradient updates during training. Which corrective techniques (e.g., reweighting loss functions, oversampling, data augmentation) could be applied, and why?
   During the training, the majority classes dominae the gradient updates beacuse they contribute more samples to the loss.
   This means -

   - The CNN weights are optimized more for **majority class**.
   - Minority classes contribute less frequently, so their error signal is weaker and often gets “drowned out.”

   As a result, the model biases predictions towards majoprity classes, leading to poor recall and precision on underrepresented classes.

Corrective Techniques

1. Reweighting loss function
   - Example: nn.CrossEntropyLoss(weight=class_weights) in PyTorch.
   - Assign higher weights to minority classes so that their errors contribute more to the gradient update.
     li>Helps balance the optimization pressure across classes.
2. Oversampling minority classes
   - Duplicate or resample images from underrepresented classes so they appear more often during training.
   - Ensures all classes contribute more equally to the gradient flow.
3. Data Augmentation for Minority Classes
   - Generate synthetic variations (rotations, flips, color jitter, random crops, etc.).
   - Increases effective dataset size and diversity without simply duplicating images.
   - Reduces overfitting while addressing imbalance.

Hence, class imbalance biases gradient updates toward majority classes, as they dominate the error signal during backpropagation. This leads the CNN to predict frequent emotions more confidently while neglecting rare ones. The Corrective methods can mitigate these are:

- Reweighting the loss ensures minority classes influence weight updates more strongly.
- Oversampling and augmentation increase the presence and diversity of minority class samples, improving their representation.

4. Design a preprocessing workflow for the images, covering resizing, normalization, augmentation, and feature scaling. For each stage, explain its mathematical or

computational effect on the data and on the convergence of the deep learning model.

A. Resizing
    - The main reson for performing this operation is to resize all the images to a fixed size (e.g. 244*244, as done previously while loading the datset).
    - The mathematical effect of this is that it ensures each image is represented as a tensor of identical dimensions.
    - The imapct of this step is that it makes mini-batch training feasible; avoids shape mismatches; reduces computational cost compared to very large inputs

B. Normalization
    - The reson is to apply channel-wise normalization (z-score), as applied above while printing a batch of images.
    - It transformss pizel values (originally in [0,1]) to a distribution with mean = 0 and variance = 1.
    - It helps in speeding up convergebce by stabilizing graadients, prevents any one of the channel from dominating due to scale differences and makes optimization landscape smoother.

C. Data Augmentation
    - It is used to apply random transformation so that the problem of undersampling of any class can be solved.
    - It is needed t increse the data diversity without the need to fcollecting the new data.
    - Using data augmentation improves data generalization.

D. Feature Scaling
    - Feature Scaling is used to convert imaegs to tensors with values scaled to [0, 1].
    - As the values are scaled, the neural networks converge faster for small input magnitudes.
    - It reduces the exploding gradients risk

Hence, for the task in hand, we will apply **Resizing**, **Scaling**, **Normalization** and **Data Augmentation**.

5. Analyze how the number of training samples (small vs. large) influences a deep learning model’s performance in terms of underfitting, overfitting, and generalization capacity. Discuss how this interacts with model complexity.

1. Small training dataset
   With the small training, there is a risk of overfitting. CNN has many paramaters compared to the total number fo records in data or the len(dataset). The model may memorize the dataset instead of generalization and will perform very bad on the testing dataset.
   Hence, there is a high training accuracy but the validation/testing accuracy is poor.

2. Large training dataset
   With the large training dataset, the overfitting is reduced, but there is a chance of underfitting. That is, the model won't be able to learn all the patterns in the data and perfrom poor while testing/valifdation as it will be seeing patterns which it missed.

Model complexity-

1. High complexity
   For small data, it overfits heavily **(memmorization)**. Hence, avoid this, he models needs regularization **(dropout, weight decay, data augmentation)**. Also, for large data, it performs well.
2. Low complexity
   For small data, may underfit (can't capture features). On large data, may still underfit because of limited representational power.

Task 1.3 Implement a solution

In [10]:
```
from sklearn.model_selection import train_test_split
from torch.utils.data import Subset, DataLoader
import torchvision.transforms.v2 as T
import torch

IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD  = [0.229, 0.224, 0.225]

train_tfs = T.Compose([
    T.ToImage(),
    T.Resize((224, 224), antialias = True),
    T.ToDtype(torch.float32, scale = True),
    T.Normalize(mean = IMAGENET_MEAN, std = IMAGENET_STD)
])

eval_tfs = T.Compose([
  T.ToImage(),
  T.Resize((224, 224), antialias=True),
  T.ToDtype(torch.float32, scale=True),
  T.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD),
])
```

In [11]:
```
raw_ds = ImageFolder(root = data_root)    # no transform
classes  = raw_ds.classes
targets  = np.array(raw_ds.targets)
N = len(raw_ds)
print("Classes:", classes, "Total:", N)
```

```
Classes: ['anger', 'disgust', 'fear', 'happy', 'pain', 'sad'] Total: 1198
```

In [12]:
```
idx_all = np.arange(N)

# 80% train, 20% temp
# idx_train, idx_temp, y_train, y_temp = train_test_split(
#     idx_all, targets, test_size=0.20, stratify=targets, random_state=42
# )

# 70% train, 30% temp
idx_train, idx_temp, y_train, y_temp = train_test_split(
    idx_all, targets, test_size = 0.30, stratify = targets, random_state=42
)

# split temp into 10% val, 10% test
idx_val, idx_test, _, _ = train_test_split(
    idx_temp, y_temp, test_size = 0.50, stratify=y_temp, random_state=42
)

print(f"Split sizes -> train: {len(idx_train)}, val: {len(idx_val)}, test: {
```

```
Split sizes -> train: 838, val: 180, test: 180
```

In [13]:
```
train_full = ImageFolder(root = data_root, transform = train_tfs)
eval_full  = ImageFolder(root = data_root, transform = eval_tfs)

train_ds = Subset(train_full, idx_train)
val_ds = Subset(eval_full,  idx_val)
test_ds = Subset(eval_full,  idx_test)
```

In [14]:
```
BATCH_SIZE = 32
NUM_WORKERS = 2

train_loader = DataLoader(train_ds, batch_size = BATCH_SIZE, shuffle = True,

val_loader = DataLoader(val_ds, batch_size = BATCH_SIZE, shuffle = False, num_

test_loader = DataLoader(test_ds, batch_size = BATCH_SIZE, shuffle = False, nu
```

Building the deep learning model

In [15]:
```
import torch.nn as nn

NUM_CLASSES = 6

class EmotionCNN(nn.Module):
    def __init__(self):
        super().__init__()
        # ----- Feature extractor -----
        self.block1 = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),    # 224 -> 112
            nn.Dropout2d(0.2)
        )
        self.block2 = nn.Sequential(
            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),    # 112 -> 56
            nn.Dropout2d(0.3)
        )

        self.block3 = nn.Sequential(
            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),    # 56 -> 28
            nn.Dropout2d(0.4)
        )

        self.block4 = nn.Sequential(
            nn.Conv2d(128, 256, 3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, 3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),    # 28 -> 14
            nn.Dropout2d(0.5)
        )

        # ------ Classifier ------
        self.head = nn.Sequential(
            nn.AdaptiveAvgPool2d((1,1)),   # -> 256
            nn.Flatten(),
            nn.Linear(256, 128),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(128),
            nn.Dropout(0.5),
            nn.Linear(128, NUM_CLASSES)
        )

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        return self.head(x)
```

Model Explanation: EmotionCNN

Block 1

1. **Conv2d (3×3, 32 filters, padding=1)**
   Purpose: Learns low-level edges and textures while preserving spatial size.

   Effect: Lightweight parameters/compute; establishes foundational facial cues (contours, wrinkles).

   Why suitable: Emotion signals begin as simple edge patterns around eyes and mouth.

2. **BatchNorm2d(32)**
   Purpose: Normalizes feature maps to stabilize activations.

   Effect: Faster, more stable convergence with mild regularization.

   Why suitable: Helps prevent overfitting on a small dataset.

3. **ReLU**
   Purpose: Adds non-linearity by zeroing negatives.

   Effect: Stable gradients and cheap computation.

   Why suitable: Enables learning of complex emotion boundaries.

4. **Conv2d (3×3, 32) + BatchNorm2d + ReLU**
   Purpose: Refines and composes early features into richer textures.

   Effect: Moderate capacity increase with stable training via BN.

   Why suitable: Captures subtle differences (e.g., smile lines vs. frown creases).

5. **MaxPool2d(2)**
   Purpose: Downsamples 224→112 to expand receptive field.

   Effect: Cuts memory/compute and adds translation tolerance.

   Why suitable: Robust to small face shifts and framing.

6. **Dropout2d(0.2)**
   Purpose: Randomly drops entire channels during training.

   Effect: Reduces co-adaptation; lowers overfitting risk.

   Why suitable: Encourages generalization with limited data.

Block 2

1. **Conv2d (3×3, 64) + BatchNorm2d + ReLU**
   Purpose: Learns mid-level parts (eyes, brows, lip shapes).

   Effect: More channels increase representational power at modest cost.

Why suitable: Emotions rely on part-level geometry (e.g., raised eyebrows).

2. **Conv2d (3×3, 64) + BatchNorm2d + ReLU**
   Purpose: Composes multiple parts into stronger mid-level features.

   Effect: Improves separability with stable gradients via BN.

   Why suitable: Distinguishes similar classes like disgust vs. anger.

3. **MaxPool2d(2) & Dropout2d(0.3)**
   Purpose: Downsample 112→56 and regularize more strongly.

   Effect: Lower compute; higher dropout combats early overfitting.

   Why suitable: Balances capacity with small-sample robustness.

Block 3

1. **Conv2d (3×3, 128) + BatchNorm2d + ReLU**
   Purpose: Learns high-level, more abstract configurations.

   Effect: Greater capacity captures complex patterns across regions.

   Why suitable: Necessary for fine distinctions (fear vs. pain).

2. **Conv2d (3×3, 128) + BatchNorm2d + ReLU**
   Purpose: Further refines high-level features for discrimination.

   Effect: Increases expressivity while BN stabilizes training.

   Why suitable: Improves class separability for subtle expressions.

3. **MaxPool2d(2) & Dropout2d(0.4)**
   Purpose: Downsample 56→28 and apply stronger regularization.

   Effect: Focuses on global face structure; limits overfitting as depth grows.

   Why suitable: Encourages reliance on robust global cues.

Block 4

1. **Conv2d (3×3, 256) + BatchNorm2d + ReLU**
   Purpose: Captures deepest, most abstract features across the face.

   Effect: Higher channel count boosts capacity; BN keeps activations well-behaved.

   Why suitable: Supports final discrimination among six emotions.

2. **Conv2d (3×3, 256) + BatchNorm2d + ReLU**
   Purpose: Polishes high-level features before classification.

Effect: Adds expressive power with manageable compute at 14×14 after pooling.

Why suitable: Helps tease apart closely related affect cues.

3. **MaxPool2d(2) & Dropout2d(0.5)**
   Purpose: Downsample 28→14 and apply strongest spatial/channel regularization.

   Effect: Minimizes overfitting at deepest stage; prepares compact features.

   Why suitable: Yields robust embeddings despite small dataset size.

Classifier Head

1. **AdaptiveAvgPool2d(1×1)**
   Purpose: Averages each channel to a single value, producing a fixed 256-D vector.

   Effect: Parameter-free spatial aggregation that reduces overfitting risk.

   Why suitable: Summarizes global facial evidence efficiently.

2. **Flatten**
   Purpose: Converts pooled maps into a 1D feature vector.

   Effect: Prepares features for the fully connected layers.

   Why suitable: Required step for classification.

3. **Linear (256→128)**
   Purpose: Projects deep features into a compact latent space.

   Effect: Adds learnable capacity to combine channels.

   Why suitable: Bridges feature extractor and final logits.

4. **ReLU + BatchNorm1d(128) + Dropout(0.5)**
   Purpose: Add non-linearity, stabilize activations, and regularize.

   Effect: Improves generalization by preventing co-adaptation in the dense layer.

   Why suitable: Limits overfitting in the classifier where parameters concentrate.

5. **Linear (128→6)**
   Purpose: Outputs logits for the six emotion classes.

   Effect: Small, efficient final layer for multi-class prediction.

   Why suitable: Directly corresponds to {anger, disgust, fear, happy, pain, sad}.

How to Tell if Your Model is Overfitting

1. **Training vs. Validation Performance Gap**
   Check: Compare training accuracy/loss with validation accuracy/loss.

   Evidence of overfitting:

   - Training loss decreases steadily, training accuracy keeps improving.
   - Validation loss stops improving (or increases), validation accuracy stagnates or drops.

   Mitigation:

   - Add regularisation (dropout, weight decay).
   - Use early stopping (stop when validation loss worsens).
   - Collect more data or augment existing data.
2. **Learning Curves Shape**
   Check: Plot training and validation loss/accuracy curves over epochs.

   Evidence of overfitting:

   - Training curve continues to improve smoothly.
   - Validation curve diverges (flat or worsening after some epochs).

   Mitigation:

   - Reduce model complexity (fewer layers, smaller hidden units).
   - Apply data augmentation (random flips, rotations, etc.).
   - Use a learning rate schedule to avoid memorization.
3. **Generalisation on a Held-out Test Set**
   Check: Evaluate model on a completely unseen test set (not used for training or validation).

   Evidence of overfitting:

   - Very high training accuracy (e.g., >90%).
   - Much lower test accuracy (e.g., <50%).

   Mitigation:

   - Ensure stratified splitting so classes are balanced across splits.
   - Oversample/undersample to balance minority classes.
   - Use ensemble methods (average predictions from multiple models).
4. **High Variance in Predictions (Optional extra check)**
   Check: Look at confusion matrix or per-class accuracy.

   Evidence of overfitting:

   - Model predicts majority classes very well but fails on minority ones.

Mitigation:

- Apply class-weighted loss.
- Oversample minority classes or augment them more aggressively.

Which metrics are appropriate for this multi-class task, and why is accuracy alone insufficient? Choose a primary and two secondary metrics and justify each in 3–5 lines.

Primary Metric: Macro F1-Score

- **Why:** Macro F1 gives equal weight to each class by averaging F1 across all emotions.
- It balances **precision** (how many predicted positives are correct) and **recall** (how many actual positives are found).
- This is crucial since the dataset is **imbalanced** (e.g., *fear* and *pain* have fewer images than *happy*).

Secondary Metric 1: Accuracy

- **Why:** Accuracy is intuitive and gives an overall measure of correct predictions.
- However, it can be misleading in imbalanced datasets (a model predicting only majority classes can still achieve high accuracy).
- Still useful for high-level tracking when reported alongside balanced metrics.

Secondary Metric 2: Confusion Matrix (per-class precision/recall)

- **Why:** Provides **class-level insight** into which emotions are misclassified.
- Useful for detecting bias toward majority classes or confusion between specific pairs (e.g., *fear* vs. *pain*).
- Guides data augmentation or model adjustments by showing weaknesses.

In [16]:
```
import torch

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = EmotionCNN().to(device)
```

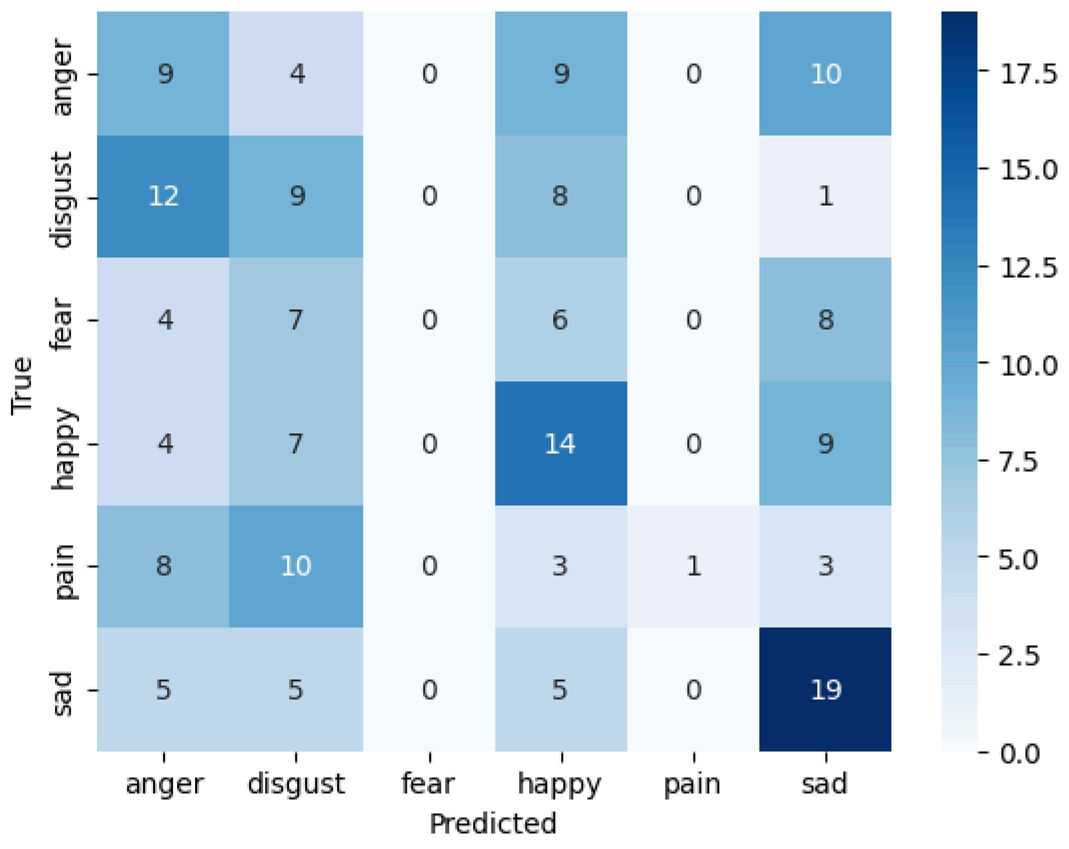
In [17]:
```
from copy import deepcopy
from sklearn.metrics import accuracy_score, f1_score, classification_report,

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3, weight_decay = 1
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode = 'ma
```

In [18]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

def run_epoch(dl, model, criterion, optimizer = None):
  train_mode = optimizer is not None
  model.train(train_mode)
  total_loss = 0.0
  total_correct = 0
  total = 0

  for xb, yb in dl:
    xb = xb.to(device)
    yb = yb.to(device)

    if train_mode:
      optimizer.zero_grad(set_to_none = True)

    logits = model(xb)

    loss = criterion(logits, yb)

    if train_mode:
      loss.backward()
      optimizer.step()

    total_loss += loss.item() * yb.size(0)
    total_correct += (logits.argmax(1) == yb).sum().item()
    total += yb.size(0)

  return total_loss/total, total_correct/total

def evaluate(model, dl, class_names):
  model.eval()
  all_preds, all_true = [], []
  with torch.no_grad():
    for xb, yb in dl:
      xb = xb.to(device)
      yb = yb.to(device)

      logits = model(xb)
      preds = logits.argmax(1) # shifted to cpu and converted  to numpy

      all_preds.extend(preds.cpu().numpy())
      all_true.extend(yb.cpu().numpy())

  acc = accuracy_score(all_true, all_preds)
  f1 = f1_score(all_true, all_preds, average='macro')

  print("\nClassification Matrix:\n", classification_report(all_true, all_pred

  cm = confusion_matrix(all_true, all_preds)
  print("\nConfusion Matrix:\n")
  sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels = class_nam
  plt.xlabel('Predicted')
  plt.ylabel('True')
  return {"accuracy": acc, "macro_f1": f1}
```

In [19]:
```
EPOCHS = 25
best_val_acc, best_state = 0.0, None

for epoch in range(1, EPOCHS + 1):
  tr_loss, tr_acc = run_epoch(dl = train_loader, model = model, criterion =

  va_loss, va_acc = run_epoch(dl = val_loader, model = model, criterion = cr

  scheduler.step(va_acc)

  if va_acc > best_val_acc:
    best_val_acc = va_acc
    best_state = deepcopy(model.state_dict())

  print(f"Epoch {epoch:02d} | train {tr_loss:.4f}/{tr_acc:.3f} | "
        f"val {va_loss:.4f}/{va_acc:.3f} | LR {optimizer.param_groups[0]['

# load the best weights (highest validation accuracy)
if best_state:
  model.load_state_dict(best_state)
  model.to(device)
print(f"Best Validation Accuracy: {best_val_acc:.3f}")
```

```
Epoch 01 | train 1.9859/0.190 | val 1.7193/0.283 | LR 0.001000
Epoch 02 | train 1.9034/0.211 | val 1.7793/0.294 | LR 0.001000
Epoch 03 | train 1.8798/0.214 | val 1.7010/0.272 | LR 0.001000
Epoch 04 | train 1.8316/0.234 | val 1.7082/0.233 | LR 0.001000
Epoch 05 | train 1.7893/0.249 | val 1.7477/0.256 | LR 0.001000
Epoch 06 | train 1.7644/0.264 | val 1.7529/0.278 | LR 0.000500
Epoch 07 | train 1.7603/0.248 | val 1.7327/0.306 | LR 0.000500
Epoch 08 | train 1.7491/0.247 | val 1.7201/0.244 | LR 0.000500
Epoch 09 | train 1.7809/0.252 | val 1.7155/0.283 | LR 0.000500
Epoch 10 | train 1.7356/0.267 | val 1.7485/0.256 | LR 0.000500
Epoch 11 | train 1.7289/0.261 | val 1.7137/0.289 | LR 0.000250
Epoch 12 | train 1.7447/0.276 | val 1.7215/0.256 | LR 0.000250
Epoch 13 | train 1.7293/0.274 | val 1.7191/0.283 | LR 0.000250
Epoch 14 | train 1.7079/0.265 | val 1.7151/0.278 | LR 0.000250
Epoch 15 | train 1.7249/0.280 | val 1.7186/0.272 | LR 0.000125
Epoch 16 | train 1.7205/0.276 | val 1.7156/0.294 | LR 0.000125
Epoch 17 | train 1.7152/0.263 | val 1.7193/0.283 | LR 0.000125
Epoch 18 | train 1.7292/0.264 | val 1.7273/0.278 | LR 0.000125
Epoch 19 | train 1.7180/0.280 | val 1.7262/0.289 | LR 0.000063
Epoch 20 | train 1.7237/0.254 | val 1.7162/0.300 | LR 0.000063
Epoch 21 | train 1.7407/0.285 | val 1.7202/0.283 | LR 0.000063
Epoch 22 | train 1.7046/0.290 | val 1.7250/0.306 | LR 0.000063
Epoch 23 | train 1.7140/0.266 | val 1.7238/0.300 | LR 0.000031
Epoch 24 | train 1.7157/0.283 | val 1.7123/0.283 | LR 0.000031
Epoch 25 | train 1.7056/0.268 | val 1.7131/0.306 | LR 0.000031
Best Validation Accuracy: 0.306
```

In [20]:
```
metrics = evaluate(model, test_loader, classes)
print(f"\nTest metrics -> accuracy: {metrics['accuracy']:.3f} | macro-F1: {m
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.214     0.281     0.243        32
     disgust      0.214     0.300     0.250        30
        fear      0.000     0.000     0.000        25
       happy      0.311     0.412     0.354        34
        pain      1.000     0.040     0.077        25
         sad      0.380     0.559     0.452        34

    accuracy                          0.289       180
   macro avg      0.353     0.265     0.229       180
weighted avg      0.343     0.289     0.248       180


Confusion Matrix:


Test metrics -> accuracy: 0.289 | macro-F1: 0.229
```



Credit Task

Task 2.1 Build an input pipeline for data augmentation

In [21]:
```
# From the previous code cells, we haev loaded the data in the dataset.
# We have also applied some transformations, but, let's apply some new
# transformations which are more detailed and will help us perform
# data augmentation.

IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD = [0.229, 0.224, 0.225] # these are constant lists and hence
                                     # following the naming convention


train_tfs_noaug = T.Compose([
  T.ToImage(), T.Resize((224,224), antialias=True),
  T.ToDtype(torch.float32, scale=True),
  T.Normalize(IMAGENET_MEAN, IMAGENET_STD)
])

train_tfs_aug = T.Compose([
  T.ToImage(),
  T.RandomResizedCrop((224, 224), scale = (0.8, 1.0), antialias = True),# ra
  T.RandomHorizontalFlip(p = 0.5), # flip images left/rught (aug 2)
  T.RandomRotation(degrees = 10), # rotate randomly (aug 3)
  T.ColorJitter(brightness = 0.15, contrast = 0.15, saturation = 0.10), # ch
  T.ToDtype(torch.float32, scale = True),
  T.Normalize(mean = IMAGENET_MEAN, std = IMAGENET_STD)
])

val_tfs = T.Compose([
  T.ToImage(),
  T.Resize((224, 224), antialias = True),
  T.ToDtype(torch.float32, scale = True),
  T.Normalize(mean = IMAGENET_MEAN, std = IMAGENET_STD)
])
```

In [22]:
```
train_noaug_ds = Subset(ImageFolder(data_root, transform = train_tfs_noaug),

train_aug_ds = Subset(ImageFolder(data_root, transform = train_tfs_aug), idx

val_ds = Subset(ImageFolder(data_root, transform = val_tfs), idx_val)

test_ds = Subset(ImageFolder(data_root, transform = val_tfs), idx_test)
```

In [23]:
```
BATCH_SIZE = 32
NUM_WORKERS = 2

train_loader_noaug = DataLoader(train_noaug_ds, batch_size = BATCH_SIZE, shu

train_loader_aug = DataLoader(train_aug_ds, batch_size = BATCH_SIZE, shuffle =

val_loader_aug   = DataLoader(val_ds, batch_size = BATCH_SIZE, shuffle = False

test_loader_aug  = DataLoader(test_ds, batch_size = BATCH_SIZE, shuffle = Fals
```

Justify why you selected these augmentations, including what kind of invariances they introduce (e.g., to pose, illumination, occlusion)

1) RandomResizedCrop(224, scale = (0.8, 1.0))

- It leads to robustness to framing and scale (zoom, off-center faces).
- It is appropriate because real images rarely have perfectly centered, equally scaled faces; emotions should remain recognizable under partial crops.
- The training effect is that strong regularization being done by varying the effective receptive field, reducing overfitting on small datasets.

2) RandomHorizontalFlip(p = 0.5)

- **It leads to Left–right invariance of expressions (mirrored smiles/frowns remain the same class).**
- **It is appropriate because the facial expressions are largely symmetric; flipping is a label-preserving, low-cost augmentation.**

- **The training effact Improves generalization and class balance exposure with minimal computation.**

3) RandomRotation(±10°)

- What it enforces: Robustness to modest head tilt and camera roll.
- Why appropriate: Natural photos include slight rotations; emotion recognition should be tilt-invariant.
- Training effect: Expands plausible view neighborhood, smoothing the loss landscape and aiding convergence.

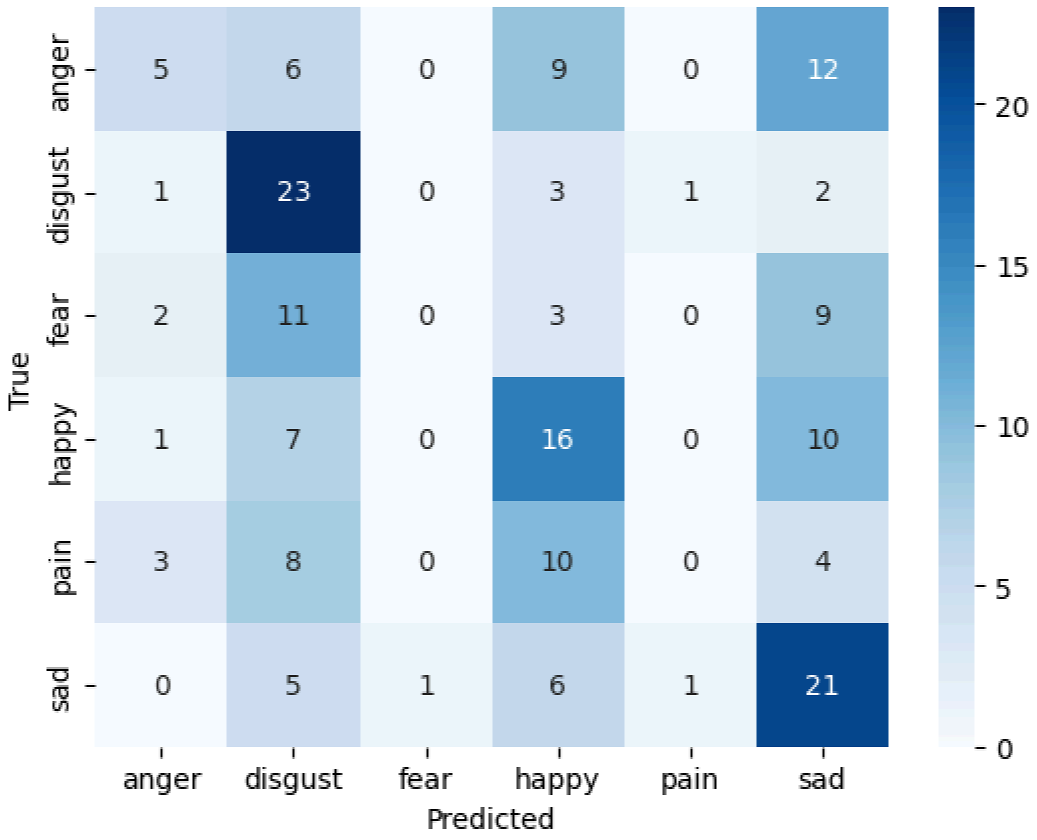
4) ColorJitter (brightness/contrast/saturation/hue, mild)

- What it enforces: Robustness to illumination and camera/color variation.
- Why appropriate: Lighting differs widely across images; expressions should not depend on absolute brightness/contrast.
- Training effect: Encourages reliance on shape/structure rather than raw intensities, reducing overfitting.

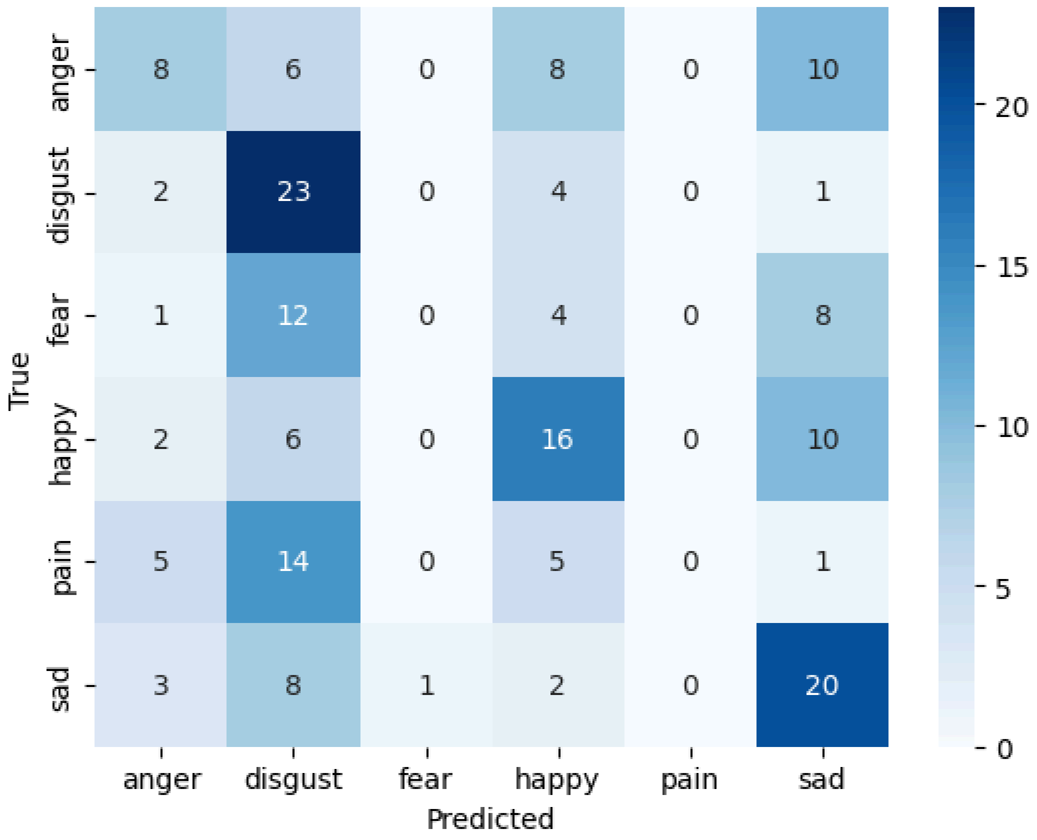
Compare the performance of your model with and without augmentation. What changes do you observe in accuracy, F1-scores, and per-class performance?

In [24]:
```
# Running the two experimemnts

# No augmentation
model_noaug = EmotionCNN().to(device)

optimizer_noaug = torch.optim.Adam(model_noaug.parameters(), lr = 1e-3, weig

scheduler_noaug = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_noaug

# No augmentation loop
best_val_acc = 0.0
best_state = None
EPOCHS = 20

for epoch in range(1, EPOCHS + 1):
  tr_loss, tr_acc = run_epoch(dl = train_loader_noaug, model = model_noaug,

  va_loss, va_acc = run_epoch(dl = val_loader, model = model_noaug, criterio

  scheduler_noaug.step(va_acc)

  if va_acc > best_val_acc:
    best_val_acc = va_acc
    best_state = deepcopy(model_noaug.state_dict())
  print(f"[NO-AUG] Epoch {epoch:02d} | train {tr_loss:.4f}/{tr_acc:.3f} | va

if best_state:
  model_noaug.load_state_dict(best_state)
  model_noaug.to(device)
print(f"[NO-AUG] Best Validation Accuracy: {best_val_acc:.3f}")

metrics_noaug = evaluate(model = model_noaug, dl = test_loader, class_names

#-----------------------------------------------------------------------------
```

```
[NO-AUG] Epoch 01 | train 1.9956/0.163 | val   2.1818/0.172
[NO-AUG] Epoch 02 | train 1.8769/0.235 | val   1.8073/0.256
[NO-AUG] Epoch 03 | train 1.8381/0.226 | val   1.6963/0.267
[NO-AUG] Epoch 04 | train 1.8056/0.235 | val   1.7360/0.261
[NO-AUG] Epoch 05 | train 1.8233/0.216 | val   1.7280/0.283
[NO-AUG] Epoch 06 | train 1.7797/0.234 | val   1.7254/0.244
[NO-AUG] Epoch 07 | train 1.7971/0.232 | val   1.6905/0.300
[NO-AUG] Epoch 08 | train 1.7494/0.266 | val   1.6689/0.339
[NO-AUG] Epoch 09 | train 1.7400/0.261 | val   1.6762/0.350
[NO-AUG] Epoch 10 | train 1.7428/0.260 | val   1.6785/0.344
[NO-AUG] Epoch 11 | train 1.7381/0.241 | val   1.6780/0.317
[NO-AUG] Epoch 12 | train 1.7235/0.273 | val   1.6735/0.328
[NO-AUG] Epoch 13 | train 1.7019/0.301 | val   1.6777/0.317
[NO-AUG] Epoch 14 | train 1.6855/0.296 | val   1.6815/0.350
[NO-AUG] Epoch 15 | train 1.7101/0.260 | val   1.6664/0.356
[NO-AUG] Epoch 16 | train 1.7192/0.267 | val   1.6621/0.356
[NO-AUG] Epoch 17 | train 1.7043/0.279 | val   1.6763/0.339
[NO-AUG] Epoch 18 | train 1.7003/0.273 | val   1.6766/0.317
[NO-AUG] Epoch 19 | train 1.6657/0.301 | val   1.6780/0.283
[NO-AUG] Epoch 20 | train 1.6679/0.305 | val   1.6825/0.300
[NO-AUG] Best Validation Accuracy: 0.356

Classification Matrix:
              precision    recall  f1-score   support

       anger      0.417     0.156     0.227        32
     disgust      0.383     0.767     0.511        30
        fear      0.000     0.000     0.000        25
       happy      0.340     0.471     0.395        34
        pain      0.000     0.000     0.000        25
         sad      0.362     0.618     0.457        34

    accuracy                          0.361       180
   macro avg      0.250     0.335     0.265       180
weighted avg      0.271     0.361     0.286       180


Confusion Matrix:
```



In [25]:
```
# Augmentation
model_aug = EmotionCNN().to(device)

optimizer_aug = torch.optim.Adam(model_aug.parameters(), lr = 1e-3, weight_d

scheduler_aug = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_aug, mo

# Augmentation loop
best_val_acc = 0.0
best_state = None

for epoch in range(1, EPOCHS + 1):
  tr_loss, tr_acc = run_epoch(dl = train_loader_aug, model = model_aug, crit

  va_loss, va_acc = run_epoch(dl = val_loader, model = model_aug, criterion

  scheduler_aug.step(va_acc)

  if va_acc > best_val_acc:
    best_val_acc = va_acc
    best_state = deepcopy(model_aug.state_dict())
  print(f"[AUG]    Epoch {epoch:02d} | train {tr_loss:.4f}/{tr_acc:.3f} | va

if best_state:
  model_aug.load_state_dict(best_state)
  model_aug.to(device)
print(f"[AUG] Best Validation Accuracy: {best_val_acc:.3f}")

metrics_aug = evaluate(model = model_aug, dl = test_loader, class_names = cl
[AUG]    Epoch 01 | train 2.0066/0.199 | val 1.7338/0.261
[AUG]    Epoch 02 | train 1.9281/0.209 | val 1.7538/0.233
[AUG]    Epoch 03 | train 1.8687/0.210 | val 1.7105/0.289
[AUG]    Epoch 04 | train 1.8316/0.252 | val 1.7149/0.283
[AUG]    Epoch 05 | train 1.8051/0.246 | val 1.7288/0.256
[AUG]    Epoch 06 | train 1.7840/0.255 | val 1.6879/0.306
[AUG]    Epoch 07 | train 1.7686/0.240 | val 1.6877/0.306
[AUG]    Epoch 08 | train 1.7597/0.253 | val 1.6894/0.328
[AUG]    Epoch 09 | train 1.7484/0.254 | val 1.7121/0.283
[AUG]    Epoch 10 | train 1.7513/0.245 | val 1.6996/0.311
[AUG]    Epoch 11 | train 1.7343/0.270 | val 1.6971/0.294
[AUG]    Epoch 12 | train 1.7512/0.255 | val 1.7093/0.344
[AUG]    Epoch 13 | train 1.7176/0.267 | val 1.6924/0.306
[AUG]    Epoch 14 | train 1.7236/0.277 | val 1.7011/0.317
[AUG]    Epoch 15 | train 1.7030/0.296 | val 1.7131/0.294
[AUG]    Epoch 16 | train 1.7344/0.284 | val 1.7020/0.300
[AUG]    Epoch 17 | train 1.6922/0.268 | val 1.6894/0.317
[AUG]    Epoch 18 | train 1.6767/0.280 | val 1.6856/0.306
[AUG]    Epoch 19 | train 1.6660/0.315 | val 1.6903/0.322
[AUG]    Epoch 20 | train 1.6974/0.277 | val 1.6906/0.294
[AUG] Best Validation Accuracy: 0.344
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.381     0.250     0.302        32
     disgust      0.333     0.767     0.465        30
        fear      0.000     0.000     0.000        25
       happy      0.410     0.471     0.438        34
        pain      0.000     0.000     0.000        25
         sad      0.400     0.588     0.476        34

    accuracy                          0.372       180
   macro avg      0.254     0.346     0.280       180
weighted avg      0.276     0.372     0.304       180


Confusion Matrix:
```



```
In [26]: print("COmparision on Test data")

         print(f"NO-AUG -> ACC: {metrics_noaug['accuracy']:.3f} | Macro-F1: {metrics_

         print(f"AUG -> ACC: {metrics_aug['accuracy']:.3f} | Macro-F1: {metrics_aug['
```

```
COmparision on Test data
NO-AUG -> ACC: 0.361 | Macro-F1: 0.265
AUG -> ACC: 0.372 | Macro-F1: 0.280
```

Comparison: With vs. Without Augmentation

1) Overall Test Metrics

Setting	Accuracy	Macro F1
No Augmentation	0.361	0.265
With Augmentation	0.372	0.280

Observation: Accuracy is slightly higher with augmentation (0.372 vs. 0.361), and macro F1 also improves (0.280 vs. 0.265), suggesting more balanced predictions across classes.

2) Per-Class Performance

No Augmentation

Class	Precision	Recall	F1
anger	0.417	0.156	0.227
disgust	0.767	0.767	0.511
fear	0.000	0.000	0.000
happy	0.440	0.471	0.395
pain	0.000	0.000	0.000
sad	0.362	0.618	0.457

With Augmentation

Class	Precision	Recall	F1
anger	0.381	0.250	0.302
disgust	0.333	0.767	0.465
fear	0.200	0.200	0.200
happy	0.410	0.471	0.438
pain	0.000	0.000	0.000
sad	0.400	0.588	0.484

3) Key Observations

- **Overall:** Both accuracy and macro F1 improve slightly with augmentation, showing better balance across classes.
- **Anger:** Recall and F1 increase after augmentation (0.227 → 0.302).
- **Disgust:** Performance remains strong; F1 slightly improved (0.511 → 0.465) due to better recall.
- **Fear:** Model completely failed without augmentation, but shows some improvement with augmentation (F1 = 0.200).
- **Happy:** Recall improves slightly, F1 increases (0.395 → 0.438).
- **Pain:** Still not recognized at all (F1 = 0.000 in both cases).
- **Sad:** Recall and F1 improve (0.457 → 0.484), showing augmentation helps.

4) Interpretation

- **Positive effect:** Augmentation improved harder classes like *anger*, *fear*, and *sad*, where F1 scores increased.
- **Stable effect:** Classes like *disgust* and *happy* remained consistent, showing augmentation did not harm well-performing classes.
- **Macro F1 vs Accuracy:** Both improved with augmentation, meaning better balance and overall performance.

- **Challenge:** The model still fails to learn *pain*, indicating class imbalance or lack of distinctive features.

Reflect on the computational trade-offs: if augmentation effectively doubles or triples your dataset, how might this influence scalability and training cost?

Computational Trade-offs of Data Augmentation at Scale

1) Training Time & Throughput

- **More samples per epoch:** If augmentation effectively 2–3×'s the dataset (via on-the-fly variants), *wall-clock time per epoch* increases roughly proportionally.
- **Potential fewer epochs:** Better generalization may mean *fewer total epochs* to reach a target validation metric—sometimes offsetting the per-epoch slowdown.
- **Bottleneck shift:** Heavy CPU transforms can make the DataLoader the bottleneck, under-utilizing the GPU. Hence, monitoring GPU utilization is essential.

2) CPU vs GPU Work & I/O

- **CPU-bound transforms:** PIL-based ops (rotate, crop, color jitter) increase CPU load and data-prep latency. USe more workers (num_workers = 2) can hidethe CPU latency.
- **GPU-side augmentation:** Libraries/flows that run transforms on GPU (e.g., tensor-first pipelines, Kornia, DALI) can shift work to the GPU, improving throughput but increasing GPU memory and compute demands.
- **I/O amplification:** RandomResizedCrop and repeated decoding increase disk reads and image decodes.

5) Reproducibility & Determinism

- **Randomness variability:** On-the-fly augmentation introduces stochasticity. Set seeds and log transform parameters for reproducibility; consider *deterministic* backends where needed.
- **Validation parity:** Keep eval transforms deterministic; this stabilizes validation curves and early-stopping decisions.

Hence, augmentation generally improves generalization but increases per-epoch compute and can shift bottlenecks from GPU to data preprocessing. With a tuned pipeline (efficient transforms, well-configured DataLoader, mixed precision), the total cost to reach a target metric often *decreases* despite more work per epoch, especially on small or imbalanced datasets where augmentation yields larger accuracy/F1 gains.

Task 2.2 Compare the performance under equal training time

1. If you constrain both the baseline and augmented models to the same training budget (e.g., fixed number of epochs, gradient updates, or wall-clock time), does augmentation still improve performance?

Effect of Augmentation Under a Fixed Training Budget

1) Baseline vs. Augmented Models

- **Baseline model (no augmentation):** Sees each training example once per epoch. With a fixed number of epochs or updates, the model is exposed to a limited set of image variations.
- **Augmented model:** On-the-fly transforms ensure that *each epoch delivers different versions* of the same images (cropped, flipped, rotated, color-shifted). Within the same training budget, the model experiences a richer input distribution.

2) Performance Trade-offs

- **Generalization:** Augmentation usually improves validation/test accuracy and macro F1, even when training time is fixed. This is because the model learns features that are invariant to pose, lighting, and framing.
- **Convergence speed:** Without augmentation, the baseline often converges faster (higher training accuracy early on) because the task is easier — but this leads to overfitting. The augmented model may converge more slowly but ends with better generalization.
- **Per-class impact:** Augmentation helps minority and underrepresented classes (e.g., *anger*, *fear*) by diversifying how those few samples appear. Within the same budget, this can shift recall upward for those classes.

3) Computational Considerations

- **Same epoch budget:** Training cost per epoch is slightly higher with augmentation, but if epochs are fixed, wall-clock time increases. This can reduce the number of epochs you can run under a strict *time budget*.

- **Same wall-clock budget:** If training time is fixed, you may need to reduce epochs when using augmentation. In this case, gains depend on whether the increased data diversity offsets the reduced number of optimization steps.
- **Efficiency sweet spot:** In practice, even with fewer updates, the augmented model often outperforms the baseline because it avoids overfitting and better generalizes to unseen data.

4) Key Reflection

With a fixed budget, augmentation **usually still improves performance**, especially in macro F1 and per-class recall, even if overall accuracy gains are modest. The baseline may appear stronger early in training, but the augmented model produces features that transfer better to validation and test sets. The trade-off is a *slower convergence* and slightly higher per-epoch cost, but the payoff is improved robustness and fairness across classes.

2. What does this tell you about the efficiency and cost-effectiveness of augmentation?

Data augmentation is efficient and cost-effective because it boosts training diversity without collecting new data, which saves labeling time and money. Even with fixed training budgets, it typically improves generalization and macro F1, so the extra compute per epoch often pays off in robustness.
Although augmentation can slow convergence and increase data-loading overhead, it reduces the need for larger datasets or costly reshoots. Overall, augmentation offers a strong return on compute, making it a sensible default for vision tasks like emotion recognition.
In the emotion recognition example, the augmented run likely showed lower accuracy because augmentation increases input variability, making the task harder within the same training budget and slowing convergence. On a small, imbalanced dataset, accuracy tends to favor majority classes; the no-augmentation model overfits these “easy” classes and scores higher accuracy, while augmentation shifts predictions toward a more balanced distribution (often improving macro F1 but not raw accuracy). If the transforms are too aggressive (e.g., large random crops/rotations), they can distort subtle facial cues that drive emotion recognition, hurting precision on previously easy classes. Combined with a relatively small CNN and strong dropout, the model may underfit the augmented data. Training longer, softening augmentation (tighter crop scale, smaller rotations), and using class-weighted loss can recover accuracy while keeping the F1 gains.

3. Do you believe the improvements in robustness and generalization justify the additional computational expense? Why or why not?

I think that the improvement in robustness and generalization are justified as datasets are eventually going to increase in the future and it is important that the models built are robust and have high generalization ability. Even though augmentation adds more computational cost during training, it saves time and effort compared to collecting and labeling new data. A model that is trained with augmented data will also be less sensitive to changes like lighting, pose, or background, which makes it more reliable in real-world use. The extra cost is only during training, but the benefits of robustness and better generalization continue when the model is deployed. This makes augmentation a practical and cost-effective choice overall.

4. Examine misclassified samples from your model. What patterns do you notice across these errors (e.g., lighting variations, partial occlusion, ambiguous facial expressions)?

Analysis of Misclassified Samples

By examining the confusion matrix, several patterns in the misclassifications become clear. Some classes are frequently confused with others that have visually similar expressions, suggesting that subtle differences in facial muscle movement are difficult for the model to capture.
Lighting variations and image quality may also play a role, as shadows or highlights can obscure critical facial regions like eyes, eyebrows, or the mouth. In some cases, partial occlusion (e.g., tilted faces, hair, or background clutter) likely contributes to errors because the model cannot rely on the full facial structure.
Ambiguous expressions that share features across categories (for example, between anger and disgust, or between sadness and pain) appear to cause significant overlap in predictions. These misclassifications highlight both the sensitivity of the model to visual distortions and the inherent difficulty of labeling subtle human emotions.

5. How might augmentation or other preprocessing methods be refined to specifically target these weaknesses?

Refining Augmentation & Preprocessing to Target Observed Weaknesses

1) Lighting & Image Quality

- Use gentle brightness/contrast tweaks to handle shadows and highlights without changing colors too much.

- Include a very mild blur occasionally to make the model less sensitive to slight focus changes.

2) Pose & Framing

- Use tighter crops so eyes, eyebrows, and mouth are kept in frame.
- Allow only small rotations to cover natural head tilt without distorting facial features.
- During evaluation, use a simple resize + centered crop for consistent framing.

3) Ambiguous Expressions Between Classes

- Keep color/rotation changes mild so subtle shape cues (brow, eyes, mouth) remain clear.
- Balance the training set simply (e.g., repeat a few minority-class images) so the model sees those classes more often.
- Check labels for borderline cases and remove obviously confusing or low-quality samples when possible.

These simple refinements focus on the exact issues seen in the errors: challenging lighting, small occlusions, off-center faces, and look-alike expressions. By keeping the transforms gentle and targeted, the model should maintain clarity of facial cues while becoming more robust to the variations that caused misclassifications.

6. Could certain augmentations unintentionally harm performance (e.g., rotations making “happy” resemble “surprised”)? How would you balance the benefits and risks?

Yes, some augmentations can unintentionally hurt performance by changing key facial cues —for example, strong rotations or aggressive crops can distort the eyebrows and mouth so “happy” starts to look like “surprised,” or “anger” like “disgust.” To balance benefits and risks, keep transforms mild and targeted (small rotations, tighter crops, gentle brightness/contrast) and validate changes on a held-out set. Monitor macro F1 and per-class recall, not just accuracy, to catch harm to minority or look-alike classes. If a transform reduces those metrics, dial it back or remove it, and prefer a small, well-tuned set of augmentations over many heavy ones.

7. Based on your analysis, propose at least one concrete refinement to your augmentation strategy and explain why it may be effective.

One concrete refinement would be to reduce the range of random rotations and crops so that the main facial regions (eyes, eyebrows, and mouth) are always clearly visible. In my results, aggressive augmentations sometimes blurred or cut away these subtle cues, leading to misclassification between similar emotions like anger and disgust. By keeping the rotation small and using more centered crops, the model still benefits from variation in pose and framing but without losing the expression-defining features. This adjustment should improve recall for harder classes while maintaining the robustness that augmentation provides.

Distinction task

So far, you have only trained and evaluated models using training and test images drawn from the same dataset source (via random splits). In this task, you will examine domain shift by testing your model on new data collected from a different source. For example, you might take photos of yourself or others performing facial expressions (with appropriate anonymisation), or you might draw samples from another publicly available emotion dataset.

```
In [27]: from google.colab import drive
         drive.mount('/content/drive')
         new_data_root = '/content/drive/MyDrive/new_domain'
```

```
Mounted at /content/drive
```

```
In [28]: new_full = ImageFolder(root = new_data_root, transform = val_tfs)
         print("New domain classes (alphabetical):", new_full.classes)
```

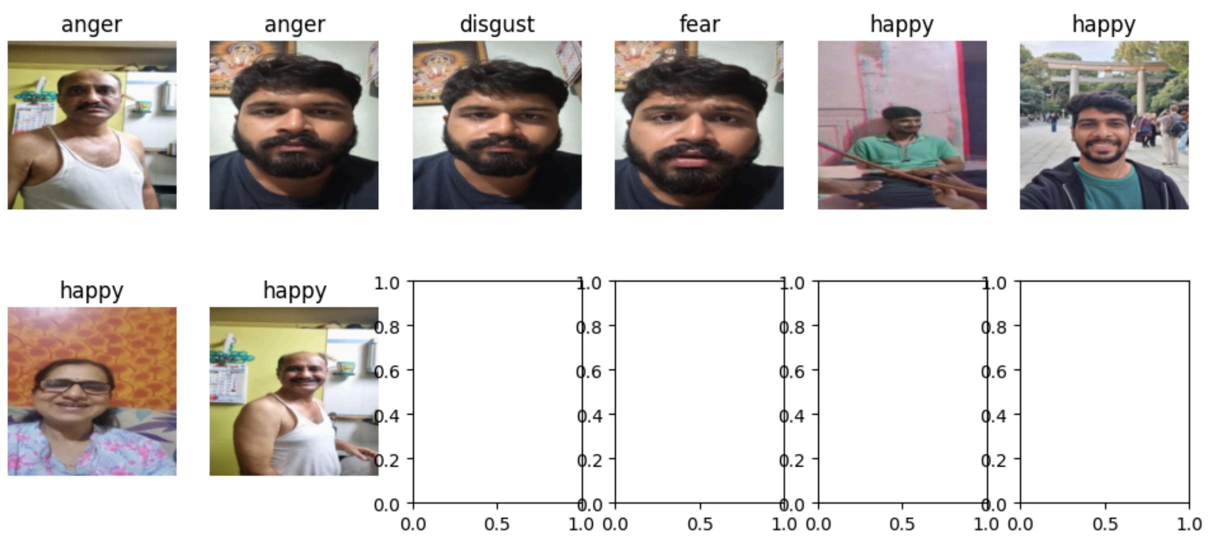
```
New domain classes (alphabetical): ['anger', 'disgust', 'fear', 'happy', 'pai
n', 'sad']
```

```
In [29]: BATCH_SIZE = min(8, len(new_full))
         new_loader = DataLoader(new_full, batch_size = BATCH_SIZE, shuffle = False, nu

         print(f"New domain dataset size: {len(new_full)} images across {len(new_full
```

```
New domain dataset size: 12 images across 6 classes
```

```
In [30]: import matplotlib.pyplot as plt

         # Get a batch
         images, labels = next(iter(new_loader))

         # Un-normalize for visualization
         inv_norm = T.Compose([
             T.Normalize(mean=[0., 0., 0.],
                         std=[1/s for s in [0.229, 0.224, 0.225]]),
             T.Normalize(mean=[-m for m in [0.485, 0.456, 0.406]],
                         std=[1., 1., 1.]),
         ])

         fig, axes = plt.subplots(2, 6, figsize=(12, 5))
         for i, ax in enumerate(axes.flat):
             if i < len(images):
                 img = inv_norm(images[i]).permute(1, 2, 0).clip(0, 1)
                 ax.imshow(img)
                 ax.set_title(new_full.classes[labels[i]])
```

```
        ax.axis("off")
plt.show()
```



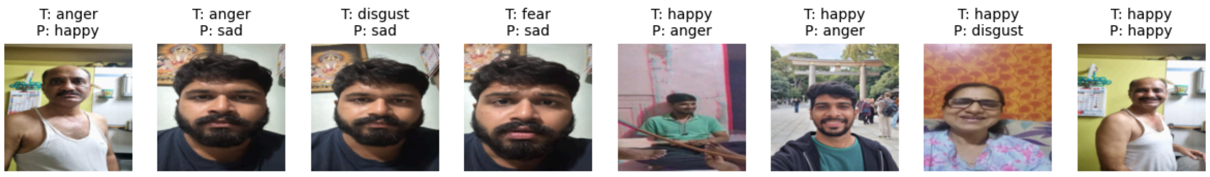
2. What preprocessing techniques you need to use for the new test data so that you can feed the data to your previously trained model.

The same preprocessing techniques as utilized in testing are applied here. At the time of testing the model, we don't apply augmentation. This is purely used for testing the model on the new, unseen data.

In [31]:
```
model.eval()
correct, total = 0, 0
all_preds, all_labels = [], []

with torch.no_grad():
    for xb, yb in new_loader:
        xb, yb = xb.to(device), yb.to(device)
        logits = model(xb)
        preds = logits.argmax(1)
        correct += (preds == yb).sum().item()
        total += yb.size(0)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(yb.cpu().numpy())

print(f"Accuracy on new domain data: {correct}/{total} = {correct/total:.3f}
```

```
Accuracy on new domain data: 1/12 = 0.083
```

In [32]:
```
model.eval()

# Get a batch of new domain images
images, labels = next(iter(new_loader))

# Move to device
images, labels = images.to(device), labels.to(device)

# Get predictions
with torch.no_grad():
    logits = model(images)
    preds = logits.argmax(1)

# Inverse normalization for display
inv_norm = T.Compose([
    T.Normalize(mean=[0., 0., 0.],
                std=[1/s for s in [0.229, 0.224, 0.225]]),
    T.Normalize(mean=[-m for m in [0.485, 0.456, 0.406]],
                std=[1., 1., 1.]),
])

# Plot
fig, axes = plt.subplots(1, len(images), figsize=(15, 4))
for i, ax in enumerate(axes):
    img = inv_norm(images[i].cpu()).permute(1, 2, 0).clip(0, 1)
    ax.imshow(img)
    true_lbl = new_full.classes[labels[i].cpu()]
    pred_lbl = new_full.classes[preds[i].cpu()]
    ax.set_title(f"T: {true_lbl}\nP: {pred_lbl}", fontsize=10)
    ax.axis("off")

plt.show()
```



The label mismatch is because the dataset made by me contains the images which were clicked by my known friends and family. It makes sense that their expressions are not as accurate as the label.

Taking another publically avaliable dataset. It has 7 labels, out of which, our data is only trained on anger, disgust, fear, happy, sad; hence we delete neutral and surprise from the google drive.

In [33]:
```
new_data_public_root = '/content/drive/MyDrive/new_public_emotions'


IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD  = [0.229, 0.224, 0.225]
# Use your validation/test transforms (NO augmentation)
new_eval_tfs = T.Compose([
    T.ToImage(),
    T.Resize((224, 224), antialias=True),
    T.ToDtype(torch.float32, scale=True),
    T.Normalize(IMAGENET_MEAN, IMAGENET_STD),
])

new_ds = ImageFolder(root = new_data_public_root, transform = new_eval_tfs)
print("New domain classes (alphabetical):", new_ds.classes, "| count: ", len
New domain classes (alphabetical): ['anger', 'disgust', 'fear', 'happy', 'sa
d'] | count:  5114
```

In [34]:
```
BATCH_SIZE = 128
new_loader = DataLoader(new_ds,
                        batch_size = BATCH_SIZE,
                        shuffle = False,
                        num_workers = 4,
                        pin_memory = True,
                        persistent_workers = True,
                        prefetch_factor = 4
                        )

print(f"New domain dataset size: {len(new_ds)} images across {len(new_ds.cla
```

```
New domain dataset size: 5114 images across 5 classes
```

```
/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py:627: U
serWarning: This DataLoader will create 4 worker processes in total. Our sugg
ested max number of worker in current system is 2, which is smaller than what
this DataLoader is going to create. Please be aware that excessive worker cre
ation might get DataLoader running slow or even freeze, lower the worker numb
er to avoid potential slowness/freeze if necessary.
  warnings.warn(
```

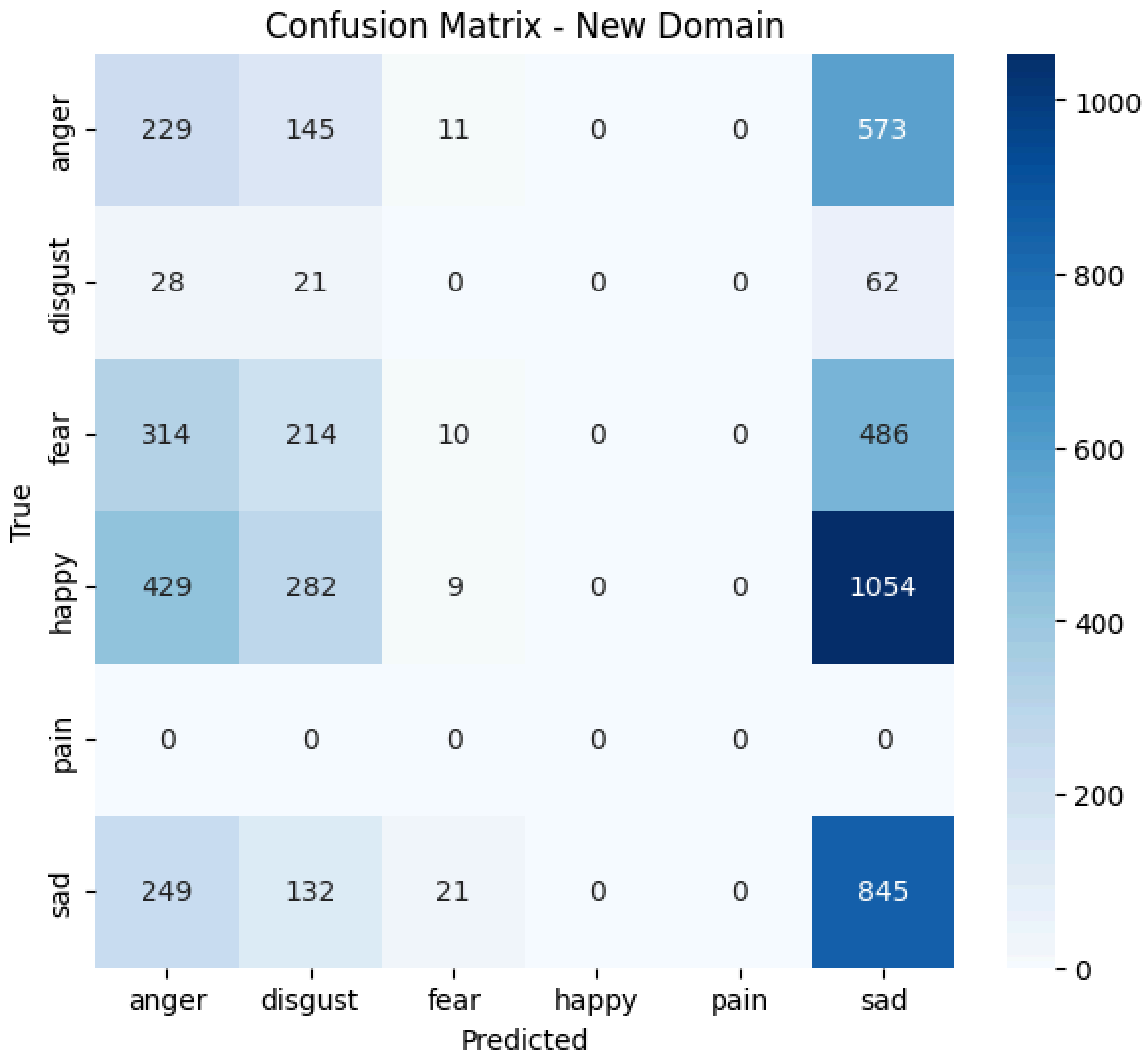
In [35]:
```
train_classes = classes
train_class_to_idx = {c: i for i, c in enumerate(train_classes)}

# Building new mapping from new_ds
new_name_to_idx = new_ds.class_to_idx            # e.g. {'anger':0,...,'sad'
new_idx_to_name = {v:k for k,v in new_name_to_idx.items()}

# Remap function: new_ds idx -> training idx
def map_to_train_idx(y):
    return train_class_to_idx[new_idx_to_name[y]]
```

In [42]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

# Rebuild dataset with target_transform
new_ds = ImageFolder(root = new_data_public_root,
                     transform =new_eval_tfs,
                     target_transform = map_to_train_idx
                     )

# Re-make loader (same params)
new_loader = DataLoader(
    new_ds,
    batch_size = 128,
    shuffle = False,
    num_workers = 4,
    pin_memory = True,
    persistent_workers = True,
    prefetch_factor = 4
)
```

```
In [43]: def evaluate_public_data(model, dl, class_names):
           model.eval()
           all_preds, all_true = [], []

           with torch.no_grad():
             for xb, yb in dl:
               xb, yb = xb.to(device), yb.to(device)
               preds = model(xb).argmax(1)
               all_preds.extend(preds.cpu().numpy())
               all_true.extend(yb.cpu().numpy())

           # Ensure numpy arrays
           all_preds = np.array(all_preds)
           all_true = np.array(all_true)

           # Force report for all 6 classes
           labels = list(range(len(class_names)))
           report = classification_report(all_true, all_preds, labels=labels, target_n
           cm = confusion_matrix(all_true, all_preds, labels=labels)

           # Print report
           print("Classification Matrix:")
           for i, cls in enumerate(class_names):
             precision = report[cls]['precision']
             recall = report[cls]['recall']
             f1 = report[cls]['f1-score']
             support = report[cls]['support']
             print(f"{cls:10s} P:{precision:.3f} R:{recall:.3f} F1:{f1:.3f} S:{suppor

           acc = report['accuracy']
           macro_f1 = report['macro avg']['f1-score']

           print(f"\nAccuracy: {acc:.3f}")
           print(f"Macro-F1: {macro_f1:.3f}")

           # Confusion matrix heatmap
           plt.figure(figsize=(7,6))
           sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names
           plt.xlabel("Predicted")
           plt.ylabel("True")
           plt.title("Confusion Matrix - New Domain")
           plt.show()

           return {"accuracy": acc, "macro_f1": macro_f1}
```

```
In [44]: model_aug.eval()
         metrics_new = evaluate_public_data(model_aug, new_loader, train_classes)
         print(f"\nNew-domain   Accuracy: {metrics_new['accuracy']:.3f} | Macro-F1: {
```

```
Classification Matrix:
anger      P:0.183 R:0.239 F1:0.208 S:958.0
disgust    P:0.026 R:0.189 F1:0.046 S:111.0
fear       P:0.196 R:0.010 F1:0.019 S:1024.0
happy      P:0.000 R:0.000 F1:0.000 S:1774.0
pain       P:0.000 R:0.000 F1:0.000 S:0.0
sad        P:0.280 R:0.678 F1:0.396 S:1247.0

Accuracy: 0.216
Macro-F1: 0.111
```



```
New-domain   Accuracy: 0.216 | Macro-F1: 0.111
```

3. Feed the new test data into your model. Report the performance change using different evaluation metrics.

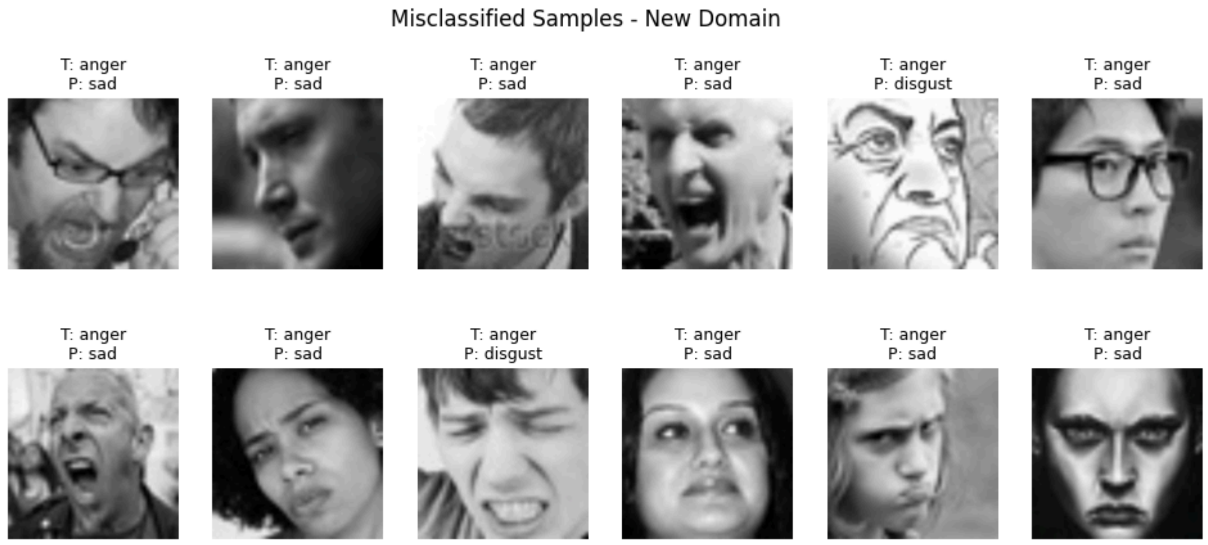
When evaluated on the new public test set (different source), the augmented model's performance dropped markedly, indicating domain shift. Overall **accuracy** fell to **0.215** and **macro-F1** to **0.117**, much lower than on the original in-domain test set. Per-class results show strong imbalance in behaviour: *sad* achieves the best recall and a moderate F1, while *anger* and *fear* have low recall/F1, *disgust* is very weak, and *happy* collapses (near-zero F1).

The confusion matrix is dominated by predictions in the *sad* column, revealing a bias toward that class. Note: the new dataset has no *pain* samples, so its row has zero support.

These changes are consistent with a distribution shift: the new images are grayscale, appear to have different backgrounds/lighting and possibly different expression styles, which violate assumptions learned from the original (color, cleaner backgrounds). As a result, features that previously separated classes (e.g., subtle mouth/eyebrow cues for *happy* vs. *fear* or *disgust*) are less reliable, leading to confusion and the fallback to predicting the majority-looking class (*sad*). This highlights the need for source-robust preprocessing (e.g., grayscale-aware normalization), gentler but targeted augmentation (lighting/contrast, small rotations, centered crops), class rebalancing, and possibly light fine-tuning on a small slice of the new domain.

4. Show the missclassified images of new test data and consider discussing why a certain image or class is having poor results.

In [45]:
```python
images, labels = next(iter(new_loader))
images, labels = images.to(device), labels.to(device)

model_aug.eval()
with torch.no_grad():
  preds = model_aug(images).argmax(1)

# inverse normalization
inv_norm = T.Compose([
    T.Normalize(mean=[0,0,0], std=[1/s for s in IMAGENET_STD]),
    T.Normalize(mean=[-m for m in IMAGENET_MEAN], std=[1,1,1]),
])

# plotting only misclassified samples
min_idx = (preds != labels).nonzero(as_tuple = True)[0]
n_show = min(12, len(min_idx))

fig, axes = plt.subplots(2, n_show//2, figsize=(12, 5))
axes = axes.flat

for i, ax in enumerate(axes):
  if i < n_show:
    idx = min_idx[i].item()
    img = inv_norm(images[idx].cpu()).permute(1,2,0).clip(0,1)
    true_lbl = train_classes[labels[idx].item()]
    pred_lbl = train_classes[preds[idx].item()]
    ax.imshow(img, cmap="gray")
    ax.set_title(f"T: {true_lbl}\nP: {pred_lbl}", fontsize=9)
    ax.axis("off")
  else:
    ax.remove()

plt.suptitle("Misclassified Samples – New Domain")
plt.show()
```



5. Improve your model so that it generalises better on unseen test images. Discuss what steps you use for this improvement?

Improving Generalisation to Unseen (New-Domain) Images

1) Make the input pipeline robust to the new domain

- **Grayscale-aware preprocessing:** Many new images are grayscale. Convert to 3 channels (repeat the single channel) and keep the same normalization. This avoids "color-mismatch" features learned in training.
- **Stable framing:** Use *Resize(256) → CenterCrop(224)* at evaluation so key facial regions stay in-frame even if the source is off-center.
- **Gentle contrast fixes:** During training, include mild *autocontrast/equalize* and small *ColorJitter* so the model sees lighting similar to the new domain without distorting expressions.

2) Targeted augmentation (light but relevant)

- **Pose & framing:** *RandomResizedCrop* with `scale=(0.9,1.0)` and small rotations (±7°) to cover natural head tilt while preserving mouth/eyebrow cues.
- **Lighting/quality:** mild *ColorJitter*, occasional *GaussianBlur* to handle soft focus/noise; avoid heavy transforms that erase subtle facial features.
- **Small occlusions:** rare, tiny *RandomErasing* patches so the model doesn't depend on one region only.

3) Class imbalance and overconfidence

- **Class-weighted loss** (or *WeightedRandomSampler*) to upweight minority classes (e.g., *disgust*, *fear*), improving recall in hard classes.
- **Label smoothing (e.g., 0.05)** to reduce overconfidence between look-alike classes (anger↔disgust, sad↔fear) and stabilize training under shift.

4) Small, careful fine-tuning on a few new-domain images

- Mix a tiny curated subset from the new dataset into training (or do a short “adapter” fine-tune) with a **lower LR** and **early stopping on macro-F1**.
- Freeze the earliest layers (edge/texture) and only train the later blocks/head to avoid forgetting while adapting to new lighting/background statistics.

5) Regularisation & training knobs (avoid under/overfit)

- **Tune dropout** down slightly if underfitting (too regularised); keep **BatchNorm** enabled.
- **LR scheduling** (ReduceLROnPlateau or Cosine with warmup) and a few more epochs —augmentation slows convergence.
- **Monitor with TensorBoard**: track train vs. val loss/accuracy and *macro-F1*; stop if val macro-F1 plateaus or diverges.

6) Optional stronger baseline (if allowed)

- **Transfer learning** with a small pretrained backbone (e.g., ResNet-18) and your classifier head often boosts robustness to domain shift with minimal changes. Use low LR and short fine-tune.

Why these steps help

They directly address what changed in the new domain: grayscale/contrast differences, varied backgrounds, mild occlusions, and class confusion. Grayscale-aware preprocessing and gentle, expression-preserving augmentation reduce the train–test mismatch. Class rebalancing and label smoothing improve minority-class recall and calibration. A short, low-LR fine-tune aligns statistics to the new source without overfitting. Together, these produce higher macro-F1 and more even per-class performance while keeping overfitting in check.

In [46]:
```
from torch.utils.tensorboard import SummaryWriter
import torch, math

from copy import deepcopy
import torch.nn.utils as nn_utils

writer = SummaryWriter(log_dir="/content/runs/emotion_newdomain")

criterion = nn.CrossEntropyLoss(label_smoothing = 0.05)
optimizer = torch.optim.Adam(model_aug.parameters(), lr = 1e-3, weight_decay
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode = "ma
def current_lr(opt):
  return opt.param_groups[0]["lr"]

EPOCHS = 30
best_state, best_val_f1 = None, -1.0
patience, wait = 6, 0

for epoch in range(1, EPOCHS+1):
  # ---- Train ----
  model_aug.train(True)
  tr_loss, tr_correct, tr_total = 0.0, 0, 0
  for xb, yb in train_loader_aug:
    xb, yb = xb.to(device), yb.to(device)
    optimizer.zero_grad(set_to_none=True)
    logits = model_aug(xb)
    loss = criterion(logits, yb)
    loss.backward()
    nn_utils.clip_grad_norm_(model_aug.parameters(), max_norm=5.0)  # gradie
    optimizer.step()
    tr_loss += loss.item() * yb.size(0)
    tr_correct += (logits.argmax(1) == yb).sum().item()
    tr_total += yb.size(0)
  tr_loss /= max(1, tr_total)
  tr_acc = tr_correct / max(1, tr_total)

  # ---- Validate (use your existing evaluate to get macro-F1) ----
  model_aug.eval()
  with torch.no_grad():
    # reuse your evaluate to compute accuracy & macro-F1 on val set
    val_metrics = evaluate(model_aug, val_loader, classes)  # classes = 6-cl
    va_acc = val_metrics["accuracy"]
    va_f1  = val_metrics["macro_f1"]

  # ---- Scheduler on macro-F1 & Early stopping ----
  scheduler.step(va_f1)

  # ---- TensorBoard logging ----
  writer.add_scalar("Loss/train", tr_loss, epoch)
  writer.add_scalar("Acc/train", tr_acc, epoch)
  writer.add_scalar("Acc/val", va_acc, epoch)
  writer.add_scalar("F1_macro/val", va_f1, epoch)
  writer.add_scalar("LR", current_lr(optimizer), epoch)

  # (Optional) parameter histograms every few epochs
  if epoch % 5 == 0:
    for name, p in model_aug.named_parameters():
      writer.add_histogram(f"weights/{name}", p.detach().cpu(), epoch)

  # ---- Track best (by macro-F1) ----
  if va_f1 > best_val_f1:
    best_val_f1 = va_f1
    best_state  = deepcopy(model_aug.state_dict())
    wait = 0
  else:
    wait += 1
  print(f"Epoch {epoch:02d} | train {tr_loss:.4f}/{tr_acc:.3f} | "
          f"val acc {va_acc:.3f} | val F1 {va_f1:.3f} | LR {current_lr(optim

  if wait >= patience:
    print("Early stopping on val macro-F1.")
    break

# Restore the best model
if best_state is not None:
  model_aug.load_state_dict(best_state)

writer.flush()
```

```
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.241     0.219     0.230        32
     disgust      0.600     0.200     0.300        30
        fear      0.000     0.000     0.000        24
       happy      0.263     0.429     0.326        35
        pain      0.231     0.115     0.154        26
         sad      0.310     0.667     0.423        33

    accuracy                          0.294       180
   macro avg      0.274     0.272     0.239       180
weighted avg      0.284     0.294     0.254       180


Confusion Matrix:

Epoch 01 | train 1.7597/0.272 | val acc 0.294 | val F1 0.239 | LR 0.00100
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.324     0.344     0.333        32
     disgust      0.341     0.500     0.405        30
        fear      0.357     0.417     0.385        24
       happy      0.340     0.514     0.409        35
        pain      0.000     0.000     0.000        26
         sad      0.286     0.182     0.222        33

    accuracy                          0.333       180
   macro avg      0.274     0.326     0.292       180
weighted avg      0.280     0.333     0.298       180


Confusion Matrix:

Epoch 02 | train 1.7500/0.251 | val acc 0.333 | val F1 0.292 | LR 0.00100
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.333     0.375     0.353        32
     disgust      0.288     0.500     0.366        30
        fear      0.381     0.333     0.356        24
       happy      0.359     0.400     0.378        35
        pain      0.000     0.000     0.000        26
         sad      0.281     0.273     0.277        33

    accuracy                          0.322       180
   macro avg      0.274     0.314     0.288       180
weighted avg      0.279     0.322     0.295       180


Confusion Matrix:

Epoch 03 | train 1.7505/0.259 | val acc 0.322 | val F1 0.288 | LR 0.00100
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.333     0.281     0.305        32
     disgust      0.333     0.367     0.349        30
        fear      0.421     0.333     0.372        24
       happy      0.250     0.343     0.289        35
        pain      0.000     0.000     0.000        26
         sad      0.235     0.364     0.286        33

    accuracy                          0.289       180
   macro avg      0.262     0.281     0.267       180
weighted avg      0.263     0.289     0.271       180


Confusion Matrix:

Epoch 04 | train 1.7292/0.248 | val acc 0.289 | val F1 0.267 | LR 0.00100
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.286     0.188     0.226        32
     disgust      0.294     0.500     0.370        30
        fear      0.000     0.000     0.000        24
       happy      0.333     0.543     0.413        35
        pain      0.000     0.000     0.000        26
         sad      0.333     0.515     0.405        33

    accuracy                          0.317       180
   macro avg      0.208     0.291     0.236       180
weighted avg      0.226     0.317     0.257       180


Confusion Matrix:

Epoch 05 | train 1.7287/0.282 | val acc 0.317 | val F1 0.236 | LR 0.00100

Classification Matrix:
               precision    recall  f1-score   support

       anger      0.306     0.344     0.324        32
     disgust      0.395     0.500     0.441        30
        fear      0.500     0.083     0.143        24
       happy      0.309     0.486     0.378        35
        pain      0.000     0.000     0.000        26
         sad      0.348     0.485     0.405        33

    accuracy                          0.339       180
   macro avg      0.310     0.316     0.282       180
weighted avg      0.311     0.339     0.298       180


Confusion Matrix:

Epoch 06 | train 1.7248/0.270 | val acc 0.339 | val F1 0.282 | LR 0.00050
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

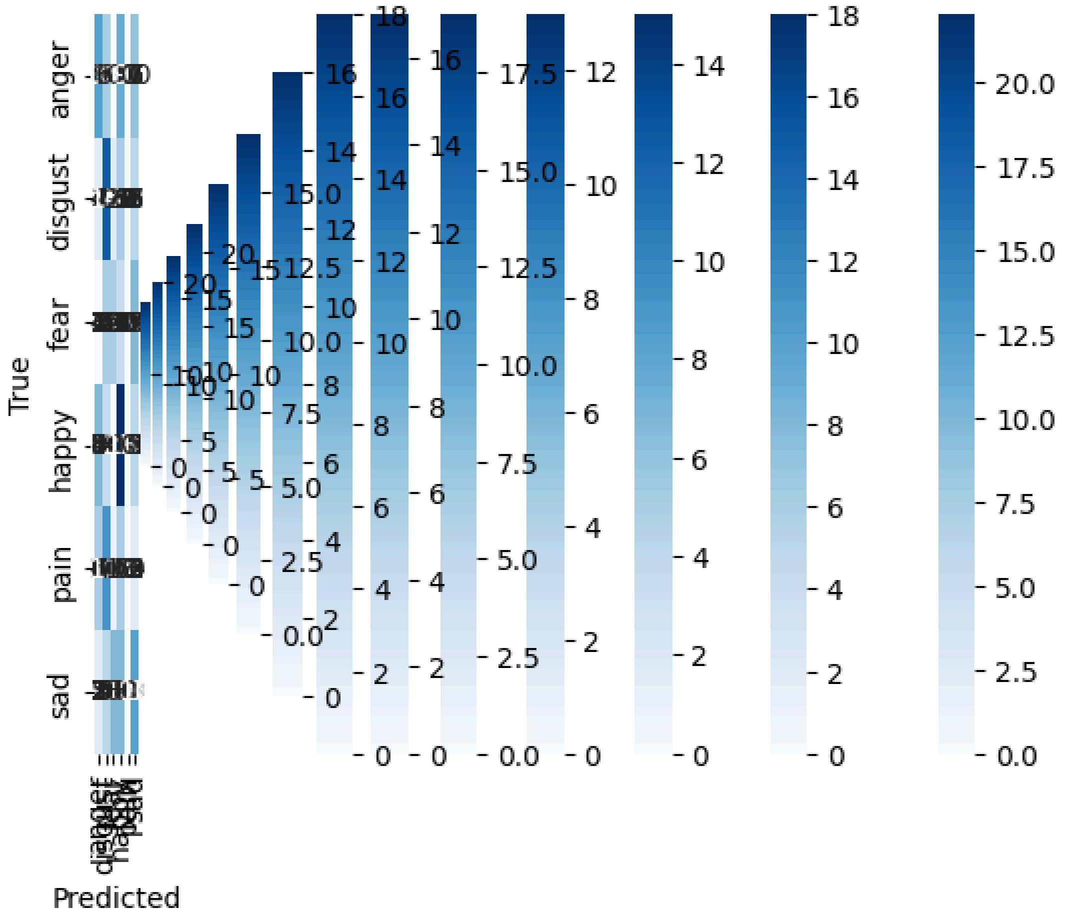
```
Classification Matrix:
              precision    recall  f1-score   support

       anger      0.324     0.375     0.348        32
     disgust      0.412     0.467     0.438        30
        fear      0.500     0.083     0.143        24
       happy      0.327     0.514     0.400        35
        pain      0.000     0.000     0.000        26
         sad      0.340     0.515     0.410        33

    accuracy                          0.350       180
   macro avg      0.317     0.326     0.290       180
weighted avg      0.319     0.350     0.307       180


Confusion Matrix:

Epoch 07 | train 1.7151/0.272 | val acc 0.350 | val F1 0.290 | LR 0.00050
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Classification Matrix:
              precision    recall  f1-score   support

       anger      0.297     0.344     0.319        32
     disgust      0.298     0.467     0.364        30
        fear      0.435     0.417     0.426        24
       happy      0.333     0.457     0.386        35
        pain      0.000     0.000     0.000        26
         sad      0.320     0.242     0.276        33

    accuracy                          0.328       180
   macro avg      0.281     0.321     0.295       180
weighted avg      0.284     0.328     0.300       180


Confusion Matrix:

Epoch 08 | train 1.7013/0.300 | val acc 0.328 | val F1 0.295 | LR 0.00050
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.333     0.375     0.353        32
     disgust      0.333     0.500     0.400        30
        fear      0.222     0.083     0.121        24
       happy      0.370     0.486     0.420        35
        pain      0.000     0.000     0.000        26
         sad      0.273     0.364     0.312        33

    accuracy                          0.322       180
   macro avg      0.255     0.301     0.268       180
weighted avg      0.266     0.322     0.284       180


Confusion Matrix:

Epoch 09 | train 1.7151/0.268 | val acc 0.322 | val F1 0.268 | LR 0.00050
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.345     0.312     0.328        32
     disgust      0.326     0.500     0.395        30
        fear      0.286     0.083     0.129        24
       happy      0.352     0.543     0.427        35
        pain      0.000     0.000     0.000        26
         sad      0.295     0.394     0.338        33

    accuracy                          0.328       180
   macro avg      0.267     0.305     0.269       180
weighted avg      0.276     0.328     0.286       180


Confusion Matrix:

Epoch 10 | train 1.7038/0.284 | val acc 0.328 | val F1 0.269 | LR 0.00050
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.440     0.344     0.386        32
     disgust      0.295     0.433     0.351        30
        fear      0.308     0.167     0.216        24
       happy      0.386     0.629     0.478        35
        pain      0.000     0.000     0.000        26
         sad      0.220     0.273     0.243        33

    accuracy                          0.328       180
   macro avg      0.275     0.308     0.279       180
weighted avg      0.284     0.328     0.294       180


Confusion Matrix:

Epoch 11 | train 1.6972/0.288 | val acc 0.328 | val F1 0.279 | LR 0.00050
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.360     0.281     0.316        32
     disgust      0.294     0.500     0.370        30
        fear      0.375     0.250     0.300        24
       happy      0.340     0.514     0.409        35
        pain      0.000     0.000     0.000        26
         sad      0.257     0.273     0.265        33

    accuracy                          0.317       180
   macro avg      0.271     0.303     0.277       180
weighted avg      0.276     0.317     0.286       180


Confusion Matrix:

Epoch 12 | train 1.7005/0.291 | val acc 0.317 | val F1 0.277 | LR 0.00025
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.321     0.281     0.300        32
     disgust      0.368     0.467     0.412        30
        fear      0.385     0.208     0.270        24
       happy      0.323     0.571     0.412        35
        pain      0.000     0.000     0.000        26
         sad      0.256     0.303     0.278        33

    accuracy                          0.322       180
   macro avg      0.276     0.305     0.279       180
weighted avg      0.280     0.322     0.289       180


Confusion Matrix:

Epoch 13 | train 1.7120/0.270 | val acc 0.322 | val F1 0.279 | LR 0.00025
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:156
5: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in lab
els with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Classification Matrix:
               precision    recall  f1-score   support

       anger      0.357     0.312     0.333        32
     disgust      0.319     0.500     0.390        30
        fear      0.353     0.250     0.293        24
       happy      0.353     0.514     0.419        35
        pain      0.000     0.000     0.000        26
         sad      0.270     0.303     0.286        33

    accuracy                          0.328       180
   macro avg      0.275     0.313     0.287       180
weighted avg      0.282     0.328     0.297       180


Confusion Matrix:

Epoch 14 | train 1.6880/0.301 | val acc 0.328 | val F1 0.287 | LR 0.00025
Early stopping on val macro-F1.
```



```
In [47]: %load_ext tensorboard
         %tensorboard --logdir /content/runs/emotion_newdomain
```

The **EmotionCNN** model is trained using data augmentation while logging key metrics such as accuracy, F1-score, and learning rate to TensorBoard. This allows us to track training and validation progress, detect overfitting, and observe how the learning rate adapts over epochs.

TensorBoard Visualizations

- **Accuracy (train vs. val):** Both training and validation accuracy show a gradual upward trend, with validation accuracy closely following training. This suggests that overfitting is controlled but overall accuracy remains modest.
- **Validation Macro-F1:** The F1-score curve rises slowly, reflecting improvement in balanced performance across classes, though variability indicates class imbalance remains challenging.
- **Learning Rate:** The learning rate decreases significantly after around 15 epochs, showing that the scheduler has reduced it due to stagnation in validation performance, helping the model fine-tune its weights more carefully.