Open in Colab

Facial Expression Recognition Competition (30%)

For this competition, we will use the a facial classification dataset. The data consists of 48x48 pixel grayscale images of faces. The faces have been automatically registered so that the face is more or less centred and occupies about the same amount of space in each image.

The task is to categorize each face based on the emotion shown in the facial expression into one of seven categories (0: Angry, 1: Disgust, 2: Fear, 3: Happy, 4: Sad, 5: Surprise, 6: Neutral). The training set consists of 28,709 examples and the public test set consists of 3,589 examples.

We provide baseline code that includes the following features:

- Loding and Analysing the FER-2013 dataset using torchvision.
- Defining a simple convolutional neural network.
- How to use existing loss function for the model learning.
- Train the network on the training data.
- Test the trained network on the testing data.
- Generate prediction for the random test image(s).

The following changes could be considered:

1. Change of advanced training parameters: Learning Rate, Optimizer, Batch-size, Number of Max Epochs, and Drop-out.
2. Use of a new loss function.
3. Data augmentation
4. Architectural Changes: Batch Normalization, Residual layers, Attention Block, and other varients.

Marking Rules:

We will mark the competition based on the final test accuracy on testing images and your report.

Final mark (out of 50) = acc_mark + efficiency mark + report mark

Acc_mark 10:

We will rank all the submission results based on their test accuracy. Zero improvement over the baseline yields 0 marks. Maximum improvement over the baseline will yield 10 marks. There will be a sliding scale applied in between.

Efficiency mark 10:

Efficiency considers not only the accuracy, but the computational cost of running the model (flops: https://en.wikipedia.org/wiki/FLOPS). Efficiency for our purposes is defined to be the ratio of accuracy (in %) to Gflops. Please report the computational cost for your final model and include the efficiency calculation in your report. Maximum improvement over the baseline will yield 10 marks. Zero improvement over the baseline yields zero marks, with a sliding scale in between.

Report mark 30:

Your report should comprise:

1. An introduction showing your understanding of the task and of the baseline model: [10 marks]

2. A description of how you have modified aspects of the system to improve performance. [10 marks]

A recommended way to present a summary of this is via an "ablation study" table, eg:

Method1	Method2	Method3	Accuracy
N	N	N	60%
Y	N	N	65%
Y	Y	N	77%
Y	Y	Y	82%

3. Explanation of the methods for reducing the computational cost and/or improve the trade-off between accuracy and cost: [5 marks]

4. Limitations/Conclusions: [5 marks]

In [ ]:
```
#########################################################################################################################
### Subject: Computer Vision
### Year: 2025
### Student Name: Manh Ha Nguyen, Le Thuy An Phan
### Student ID: a1840406, a1874923
### Comptetion Name: Facial Expression Recognition/Classification
### Final Results:
### ACC: 0.6873780997492338       GFLOPs: 0.046421376
#########################################################################################################################
```

Set Up

In [4]:
```
from google.colab import drive
drive.mount('/content/drive')
```

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

In [ ]:
```
!pip install fvcore
```

```
Collecting fvcore
  Downloading fvcore-0.1.5.post20221221.tar.gz (50 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 50.2/50.2 kB 3.1 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from fvcore) (2.0.2)
Collecting yacs>=0.1.6 (from fvcore)
  Downloading yacs-0.1.8-py3-none-any.whl.metadata (639 bytes)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-packages (from fvcore) (6.0.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from fvcore) (4.67.1)
Requirement already satisfied: termcolor>=1.1 in /usr/local/lib/python3.11/dist-packages (from fvcore) (3.1.0)
Requirement already satisfied: Pillow in /usr/local/lib/python3.11/dist-packages (from fvcore) (11.2.1)
Requirement already satisfied: tabulate in /usr/local/lib/python3.11/dist-packages (from fvcore) (0.9.0)
Collecting iopath>=0.1.7 (from fvcore)
  Downloading iopath-0.1.10.tar.gz (42 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 42.2/42.2 kB 4.2 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: typing_extensions in /usr/local/lib/python3.11/dist-packages (from iopath>=0.1.7-
>fvcore) (4.13.2)
Collecting portalocker (from iopath>=0.1.7->fvcore)
  Downloading portalocker-3.1.1-py3-none-any.whl.metadata (8.6 kB)
Downloading yacs-0.1.8-py3-none-any.whl (14 kB)
Downloading portalocker-3.1.1-py3-none-any.whl (19 kB)
Building wheels for collected packages: fvcore, iopath
  Building wheel for fvcore (setup.py) ... done
  Created wheel for fvcore: filename=fvcore-0.1.5.post20221221-py3-none-any.whl size=61397 sha256=b846cee2d239ed
42a93b93360a5e1236c374f33c7615e043e9e014b6d5d1b440
  Stored in directory: /root/.cache/pip/wheels/65/71/95/3b8fde5c65c6e4a806e0867c1651dcc71a1cb2f3430e8f355f
  Building wheel for iopath (setup.py) ... done
  Created wheel for iopath: filename=iopath-0.1.10-py3-none-any.whl size=31527 sha256=1e6625e1c49e249aad5b4c448b
89b62f4b9dfd34976f092e83786d7159970664
  Stored in directory: /root/.cache/pip/wheels/ba/5e/16/6117f8fe7e9c0c161a795e10d94645ebcf301ccbd01f66d8ec
Successfully built fvcore iopath
Installing collected packages: yacs, portalocker, iopath, fvcore
Successfully installed fvcore-0.1.5.post20221221 iopath-0.1.10 portalocker-3.1.1 yacs-0.1.8
```

In [5]:
```
# Importing libraries.

import torch
import torchvision
import tarfile
import torch.nn as nn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# To avoid non-essential warnings
import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
from tqdm import tqdm
import torchvision.transforms as T
from torchvision.datasets import ImageFolder
from torchvision.transforms import ToTensor
from torchvision.utils import make_grid
from torch.utils.data import Dataset, DataLoader
```

In [6]:
```
# Mounting G-Drive to get your dataset.
# To access Google Colab GPU; Go To: Edit >>> Network Settings >>> Hardware Accelarator: Select GPU.
# Reference: https://towardsdatascience.com/google-colab-import-and-export-datasets-eccf801e2971
from google.colab import drive
drive.mount('/content/drive')
```

```
# Dataset path. Ensure that the file path correspond to the path you have here. It is expected that you unzip t
data_directory ='/content/drive/MyDrive/Datasets/fer2013/fer2013.csv'
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", forc
e_remount=True).
```

In [7]:
```
# Reading the dataset file using Pandas read_csv function and print the first
# 5 samples.
#
# Reference: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html
data_df = pd.read_csv(data_directory)
data_df.head(4)
```

Out[7]:

	emotion	pixels	Usage
0	0	70 80 82 72 58 58 60 63 54 58 60 48 89 115 121...	Training
1	0	151 150 147 155 148 133 111 140 170 174 182 15...	Training
2	2	231 212 156 164 174 138 161 173 182 200 106 38...	Training
3	4	24 32 36 30 32 23 19 20 30 41 21 22 32 34 21 1...	Training

In [8]:
```
# Mapping of the Facial Expression Labels.
Labels = {
    0:'Angry',
    1:'Disgust',
    2:'Fear',
    3:'Happy',
    4:'Sad',
    5:'Surprise',
    6:'Neutral'
}
Labels
```

Out[8]:
```
{0: 'Angry',
 1: 'Disgust',
 2: 'Fear',
 3: 'Happy',
 4: 'Sad',
 5: 'Surprise',
 6: 'Neutral'}
```

In [9]:
```
# Categorizing the dataset to three categories.
# Training: To train the model.
# PrivateTest: To test the train model; commonly known as Validation.
# PublicTest: To test the final model on Test set to check how your model perfomed. Do not use this data as you
train_df = data_df[data_df['Usage']=='Training']
valid_df = data_df[data_df['Usage']=='PublicTest']
test_df = data_df[data_df['Usage']=='PrivateTest']
print(train_df.head())
print(valid_df.head(-1))
```

```
   emotion                                             pixels     Usage
0        0  70 80 82 72 58 58 60 63 54 58 60 48 89 115 121...  Training
1        0  151 150 147 155 148 133 111 140 170 174 182 15...  Training
2        2  231 212 156 164 174 138 161 173 182 200 106 38...  Training
3        4  24 32 36 30 32 23 19 20 30 41 21 22 32 34 21 1...  Training
4        6  4 0 0 0 0 0 0 0 0 0 0 0 3 15 23 28 48 50 58 84...  Training
       emotion                                             pixels       Usage
28709        0  254 254 254 254 254 249 255 160 2 58 53 70 77 ...  PublicTest
28710        1  156 184 198 202 204 207 210 212 213 214 215 21...  PublicTest
28711        4  69 118 61 60 96 121 103 87 103 88 70 90 115 12...  PublicTest
28712        6  205 203 236 157 83 158 120 116 94 86 155 180 2...  PublicTest
28713        3  87 79 74 66 74 96 77 80 80 84 83 89 102 91 84 ...  PublicTest
...        ...                                                ...         ...
32292        3  0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 3 4 21 40 53 65 ...  PublicTest
32293        4  178 176 172 173 173 174 176 173 166 166 206 22...  PublicTest
32294        3  25 34 42 44 42 47 57 59 59 58 54 51 50 56 63 6...  PublicTest
32295        4  255 255 255 255 255 255 255 255 255 255 255 25...  PublicTest
32296        4  33 25 31 36 36 42 69 103 132 163 175 183 187 1...  PublicTest

[3588 rows x 3 columns]
```

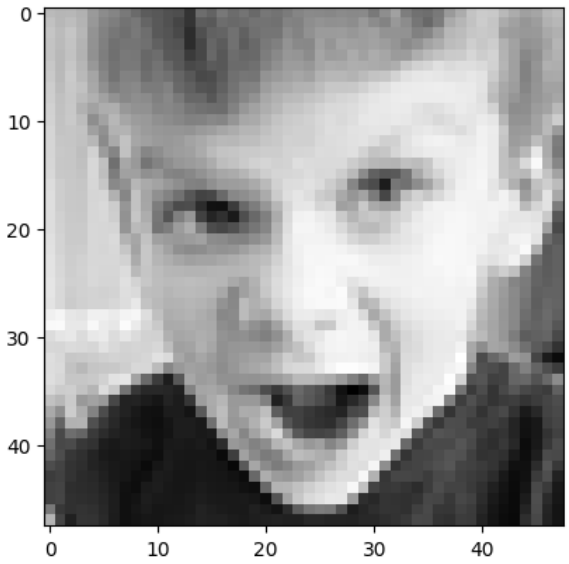
In [10]:
```
# Test-check to see wether usage labels have been allocated to the dataset/not.
valid_df = valid_df.reset_index(drop=True)
test_df = test_df.reset_index(drop=True)
print(test_df.head())
print('   -----    -------    -------    --------     -----    -------')
print(valid_df.head())
```

```
   emotion                                             pixels        Usage
0        0  170 118 101 88 88 75 78 82 66 74 68 59 63 64 6...  PrivateTest
1        5  7 5 8 6 7 3 2 6 5 4 4 5 7 5 5 5 6 7 7 7 10 10 ...  PrivateTest
2        6  232 240 241 239 237 235 246 117 24 24 22 13 12...  PrivateTest
3        4  200 197 149 139 156 89 111 58 62 95 113 117 11...  PrivateTest
4        2  40 28 33 56 45 33 31 78 152 194 200 186 196 20...  PrivateTest
   emotion                                             pixels       Usage
0        0  254 254 254 254 254 249 255 160 2 58 53 70 77 ...  PublicTest
1        1  156 184 198 202 204 207 210 212 213 214 215 21...  PublicTest
2        4  69 118 61 60 96 121 103 87 103 88 70 90 115 12...  PublicTest
3        6  205 203 236 157 83 158 120 116 94 86 155 180 2...  PublicTest
4        3  87 79 74 66 74 96 77 80 80 84 83 89 102 91 84 ...  PublicTest
```

In [11]:
```python
# Preview of the training sample and associated labels.
def show_example(df, num):
    print('expression: ' ,df.iloc[num] )
    image = np.array([[int(i) for i in x.split()] for x in df.loc[num, ['pixels']]])
    print("shape: ", image.shape)
    image = image.reshape(48,48)
    plt.imshow(image, interpolation='nearest', cmap='gray')
    plt.show()
```

In [12]:
```python
show_example(train_df, 107)
```

```
expression:  emotion                                                0
pixels     179 175 176 170 149 136 129 118 105 91 87 81 7...
Usage                                                 Training
Name: 107, dtype: object
shape:  (1, 2304)
```



In [13]:
```python
# Normalization of the train and validation data.
class expressions(Dataset):
    def __init__(self, df, transforms=None):
        self.df = df
        self.transforms = transforms

    def __len__(self):
        return len(self.df)

    def __getitem__(self, index):
        row = self.df.loc[index]
        image, label = np.array([x.split() for x in self.df.loc[index, ['pixels']]]), row['emotion']
        #image = image.reshape(1,48,48)
        image = np.asarray(image).astype(np.uint8).reshape(48,48,1)
        #image = np.reshape(image,(1,48,48))


        if self.transforms:
            image = self.transforms(image)

        return image.clone().detach(), label
```

In [14]:
```python
#import albumentations as A
stats = ([0.5],[0.5])
```

In [15]:
```python
train_tsfm = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.ToTensor(),
    T.Normalize(*stats,inplace=True),
```

```
])
valid_tsfm = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.ToTensor(),
    T.Normalize(*stats,inplace=True)
])
```

In [16]:
```
train_ds = expressions(train_df, train_tsfm)
valid_ds = expressions(valid_df, valid_tsfm)
test_ds = expressions(test_df, valid_tsfm)
```

In [17]:
```
batch_size = 400
train_dl = DataLoader(train_ds, batch_size, shuffle=True,
                      num_workers=2, pin_memory=True)
valid_dl = DataLoader(valid_ds, batch_size*2,
                    num_workers=2, pin_memory=True)
test_dl = DataLoader(test_ds, batch_size*2,
                    num_workers=2, pin_memory=True)
def show_batch(dl):
    for images, labels in dl:
        fig, ax = plt.subplots(figsize=(12, 6))
        ax.set_xticks([]); ax.set_yticks([])
        ax.imshow(make_grid(images, nrow=20).permute(1, 2, 0))
        break

show_batch(train_dl)
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-1.0..1.0].
```

In [18]:
```
# Evaluation metric - Accuracy in this case.

import torch.nn.functional as F
input_size = 48*48
output_size = len(Labels)

def accuracy(output, labels):
    predictions, preds = torch.max(output, dim=1)
    return torch.tensor(torch.sum(preds==labels).item()/len(preds))
```

In [19]:
```
# Expression model class for training and validation purpose.

class expression_model(nn.Module):

    def training_step(self, batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out, labels)
        return loss

    def validation_step(self, batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out, labels)
        acc = accuracy(out, labels)
        return {'val_loss': loss.detach(), 'val_acc': acc}
```

```
    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()
        batch_acc = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_acc).mean()
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch[{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch, result['val_loss'], result['val_acc'
```

In [20]:
```
# To check wether Google Colab GPU has been assigned/not.
torch.cuda.is_available()
def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    elif torch.backends.mps.is_available():
        return torch.device('mps')
    else:
        return torch.device('cpu')
device = get_default_device()
print(f'You are training on: {device}.')
```

```
You are training on: cuda.
```

In [21]:
```
# def to_device(data, device):
#     """Move tensor(s) to chosen device"""
#     if isinstance(data, (list,tuple)):
#         return [to_device(x, device) for x in data]
#     return data.to(device, non_blocking=True)

def to_device(data, device):
    if isinstance(data, (list, tuple)):
        return [to_device(x, device) for x in data]
    # Only use non_blocking on CUDA, not MPS
    return data.to(device, non_blocking=(device.type == 'cuda'))

class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)
```

In [22]:
```
train_dl = DeviceDataLoader(train_dl, device)
valid_dl = DeviceDataLoader(valid_dl, device)
test_dl = DeviceDataLoader(test_dl, device)
```

In [23]:
```
# Functions for evaluation and training.
import numpy as np
import scipy.stats


@torch.no_grad()
def evaluate(model, valid_dl):
    model.eval()
    outputs = [model.validation_step(batch) for batch in valid_dl]
    return model.validation_epoch_end(outputs)

# evaluate using reciporical rank fusion for multi model ensemble
@torch.no_grad()
def evaluate_rrf(models, valid_dl, weights=None):
    """
    Evaluate ensemble of models using Reciprocal Rank Fusion

    Args:
        models: List of trained models
        valid_dl: Validation dataloader
        weights: List of weights for each model (optional, defaults to equal weights)

    Returns:
        Dictionary with validation loss and accuracy
    """
    if weights is None:
        weights = [1.0] * len(models)
```

```
    # Set all models to evaluation mode
    for model in models:
        model.eval()

    # Get predictions from each model for ensemble
    all_scores = []
    all_losses = []
    true_labels = []

    for i, model in enumerate(models):
        predictions = []
        scores = []
        losses = []

        for batch in valid_dl:
            images, batch_labels = batch
            images = images.to(device)

            # Get model outputs and validation step
            val_result = model.validation_step(batch)
            losses.append(val_result['val_loss'])

            outputs = model(images)
            batch_scores = F.softmax(outputs, dim=1)
            scores.append(batch_scores.cpu().numpy())

            # Collect true labels only once
            if i == 0:
                true_labels.extend(batch_labels.cpu().numpy())

        all_scores.append(np.concatenate(scores))
        all_losses.append(torch.stack(losses).mean().item())

    true_labels = np.array(true_labels)

    # Reciprocal Rank Fusion with weights
    k = 60  # fusion parameter
    fused_scores = np.zeros_like(all_scores[0])

    for i, scores in enumerate(all_scores):
        # Convert probabilities to ranks (highest prob = rank 1)
        ranks = np.zeros_like(scores)
        for j in range(len(scores)):
            ranks[j] = len(scores[j]) - scipy.stats.rankdata(scores[j]) + 1

        # Apply weighted reciprocal rank formula
        fused_scores += weights[i] * (1 / (k + ranks))

    # Get final predictions from fused scores
    final_predictions = np.argmax(fused_scores, axis=1)

    # Calculate ensemble metrics
    ensemble_accuracy = np.mean(final_predictions == true_labels)
    ensemble_loss = np.mean(all_losses)  # Average of individual model losses

    return {
        'val_loss': ensemble_loss,
        'val_acc': ensemble_accuracy
    }

def evaluate_weighted_average_probs(models, dl, weights=None):
    """
    Evaluate ensemble using weighted average of predicted probabilities.

    Args:
        models: List of trained models
        dl: DataLoader for evaluation
        weights: List of weights for each model (if None, uses equal weights)

    Returns:
        Dictionary with validation loss and accuracy
    """
    if torch.cuda.is_available():
        device = torch.device('cuda')
    elif torch.backends.mps.is_available():
        device = torch.device('mps')
    else:
        device = torch.device('cpu')

    if weights is None:
        weights = [1.0/len(models)] * len(models)

    # Ensure weights sum to 1
    weights = np.array(weights)
    weights = weights / weights.sum()
```

```
    true_labels = []
    model_outputs_probs = []
    all_losses = []

    for i, model in enumerate(models):
        model.eval()
        model_logits_list = []
        losses = []

        with torch.no_grad():
            for batch in dl:
                images, batch_labels = batch
                images = images.to(device)

                # Get model outputs and validation step
                val_result = model.validation_step(batch)
                losses.append(val_result['val_loss'])

                outputs = model(images)
                model_logits_list.append(outputs.cpu())

                # Collect true labels only once
                if i == 0:
                    true_labels.extend(batch_labels.cpu().numpy())

        # Concatenate all logits for this model
        model_logits = torch.cat(model_logits_list, dim=0)
        model_probs = F.softmax(model_logits, dim=1)

        # Apply weight to probabilities
        weighted_probs = model_probs * weights[i]
        model_outputs_probs.append(weighted_probs)

        # Store average loss for this model
        all_losses.append(torch.stack(losses).mean().item())

    true_labels = np.array(true_labels)

    # Sum weighted probabilities from all models
    ensembled_probs = torch.stack(model_outputs_probs).sum(dim=0)
    final_preds = torch.argmax(ensembled_probs, dim=1).numpy()

    # Calculate ensemble metrics
    ensemble_accuracy = np.mean(final_preds == true_labels)
    ensemble_loss = np.mean(all_losses)  # Average of individual model losses

    return {
        'val_loss': ensemble_loss,
        'val_acc': ensemble_accuracy
    }


def fit(epochs, lr, model, train_dl, valid_dl, opt_func=torch.optim.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        # Training Phase
        model.train()
        train_losses = []
        for batch in train_dl:
            loss = model.training_step(batch)
            train_losses.append(loss)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, valid_dl)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        model.epoch_end(epoch, result)
        history.append(result)
    return history
```

In [24]:

```
# Plots for accuracy and loss during training period.
def plot_accuracies(history):
    accuracies = [x['val_acc'] for x in history]
    plt.plot(accuracies, '-x')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.title('Accuracy vs. No. of epochs');
def plot_losses(history):
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x['val_loss'] for x in history]
    plt.plot(train_losses, '-bx')
    plt.plot(val_losses, '-rx')
    plt.xlabel('epoch')
    plt.ylabel('loss')
```

```
    plt.legend(['Training', 'Validation'])
    plt.title('Loss vs. No. of epochs');
```

In [25]:
```
# Prediction function to evaluate the model.
def predict_image(img, model):
    xb = img.unsqueeze(0)
    yb = model(xb)
    _, preds  = torch.max(yb, dim=1)
    return Labels[preds[0].item()]
```

In [26]:
```
# Count number of parameters
def count_params(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

# Xavier weight initialization
def init_weights(m):
  if isinstance(m, nn.Linear):
    nn.init.xavier_uniform_(m.weight)
    if m.bias is not None:
      nn.init.zeros_(m.bias)
```

In [ ]:
```
from itertools import product

# Define weight values
def generate_weight_combinations(num_models, step=0.1, min_weight=0.1, max_weight=0.9):
    """
    Generate all weight combinations for ensemble models where weights sum to 1.0

    Args:
        num_models: Number of models in the ensemble
        step: Step size between 0 and 1 for weight values
        min_weight: Minimum weight value for any model
        max_weight: Maximum weight value for any model

    Returns:
        List of weight combinations
    """
    from itertools import product
    import numpy as np

    # Generate weight values within the specified range with given step
    weight_values = np.arange(min_weight, max_weight + step, step).round(2).tolist()

    weights = []

    # Generate all possible combinations
    for combination in product(weight_values, repeat=num_models-1):
        # Calculate the last weight to make sum = 1.0
        last_weight = round(1.0 - sum(combination), 2)

        # Check if last weight is valid (within bounds and reasonable)
        if last_weight >= min_weight and last_weight <= max_weight and abs(sum(combination) + last_weight - 1.0
            weight_combo = list(combination) + [last_weight]
            weights.append(weight_combo)

    return weights
```

Baseline Models

In [31]:
```
# Basic model - 1 layer
simple_model = nn.Sequential(
    nn.Conv2d(1, 8, kernel_size=3, stride=1, padding=1),
    nn.MaxPool2d(2, 2)
)
simple_model.to(device)
```

Out[31]:
```
Sequential(
  (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```

In [ ]:
```
for images, labels in train_dl:
    print('images.shape:', images.shape)
    out = simple_model(images)
    print('out.shape:', out.shape)
    break
```

```
images.shape: torch.Size([400, 1, 48, 48])
out.shape: torch.Size([400, 8, 24, 24])
```

In [ ]:
```
# Model - 7 layer
class expression(expression_model):
    def __init__(self,classes):
```

```
        super().__init__()
        self.num_classes = classes
        self.network = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=3, padding=1),  #(input channels, output channels)
            nn.ReLU(),
            nn.Conv2d(8, 32, kernel_size=3, padding=1),  #(input channels, output channels)
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2), # output: 64 x 24 x 24

            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2), # output: 128 x 12 x 12

            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2), # output: 256 x 6 x 6

            nn.Flatten(),
            nn.Linear(256*6*6, 2304),
            nn.ReLU(),
            nn.Linear(2304, 1152),
            nn.ReLU(),
            nn.Linear(1152, 576),
            nn.ReLU(),
            nn.Linear(576,288),
            nn.ReLU(),
            nn.Linear(288,144),
            nn.ReLU(),
            nn.Linear(144,self.num_classes))

    def forward(self, xb):
        return self.network(xb)
```

In [ ]:
```
# Model print
model = to_device(expression(classes = 7), device)
model
```

Out[ ]:
```
expression(
  (network): Sequential(
    (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(8, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU()
    (9): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU()
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU()
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU()
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Flatten(start_dim=1, end_dim=-1)
    (18): Linear(in_features=9216, out_features=2304, bias=True)
    (19): ReLU()
    (20): Linear(in_features=2304, out_features=1152, bias=True)
    (21): ReLU()
    (22): Linear(in_features=1152, out_features=576, bias=True)
    (23): ReLU()
    (24): Linear(in_features=576, out_features=288, bias=True)
    (25): ReLU()
    (26): Linear(in_features=288, out_features=144, bias=True)
    (27): ReLU()
    (28): Linear(in_features=144, out_features=7, bias=True)
  )
)
```

In [ ]:
```
evaluate(model, valid_dl)
```

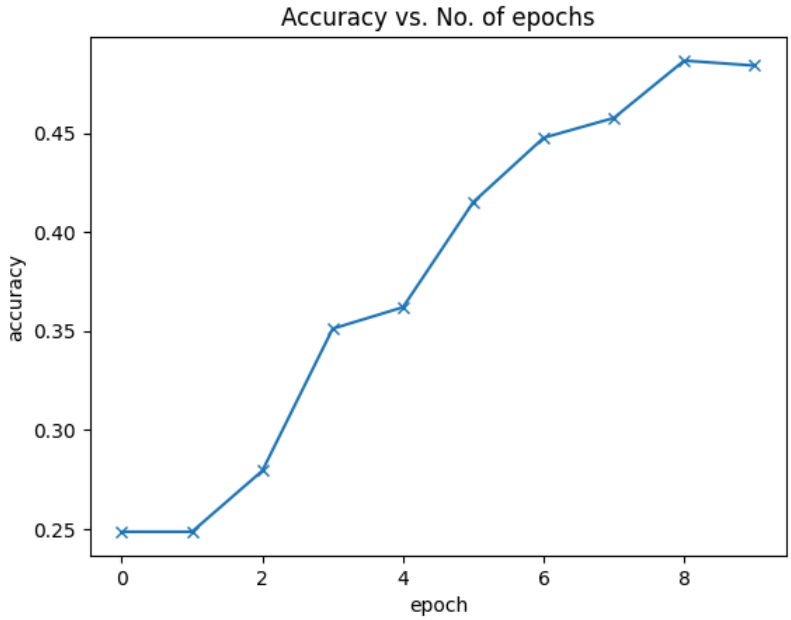
Out[ ]:
```
{'val_loss': 1.958022952079773, 'val_acc': 0.11484382301568985}
```

In [ ]:
```
num_epochs = 10
opt_func = torch.optim.Adam
lr = 0.001
```

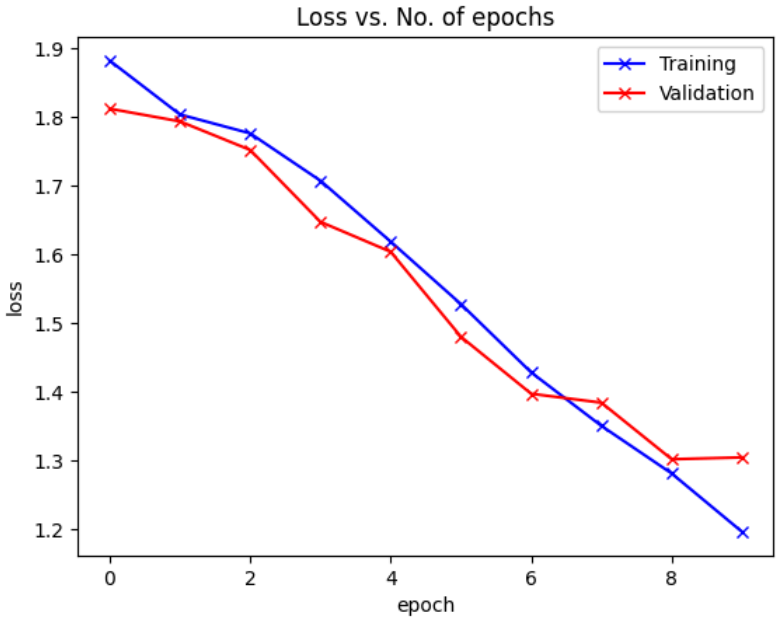
In [ ]:
```
history = fit(num_epochs, lr, model, train_dl, valid_dl, opt_func)
```

```
Epoch[0], val_loss: 1.8118, val_acc: 0.2486
Epoch[1], val_loss: 1.7933, val_acc: 0.2486
Epoch[2], val_loss: 1.7518, val_acc: 0.2794
Epoch[3], val_loss: 1.6473, val_acc: 0.3511
Epoch[4], val_loss: 1.6037, val_acc: 0.3620
Epoch[5], val_loss: 1.4802, val_acc: 0.4151
Epoch[6], val_loss: 1.3973, val_acc: 0.4476
Epoch[7], val_loss: 1.3843, val_acc: 0.4576
Epoch[8], val_loss: 1.3021, val_acc: 0.4866
Epoch[9], val_loss: 1.3048, val_acc: 0.4841
```

In [ ]:
```
plot_accuracies(history)
```



In [ ]:
```
plot_losses(history)
```

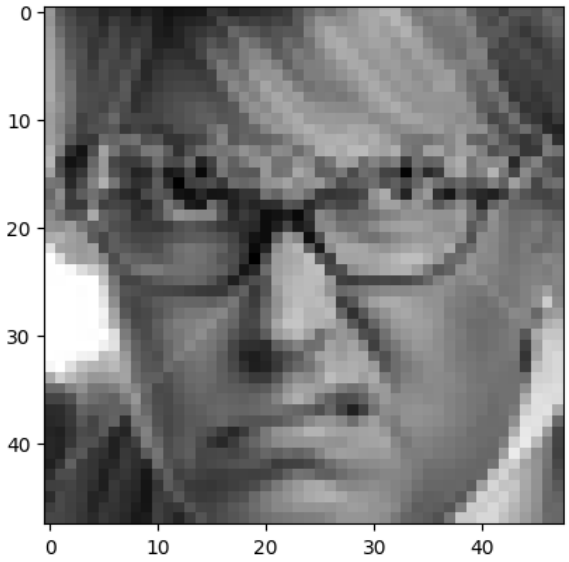


In [ ]:
```
# Model evaluation on test data.
result = evaluate(model, test_dl)
result
```

Out[ ]:
```
{'val_loss': 1.2849340438842773, 'val_acc': 0.4967525601387024}
```

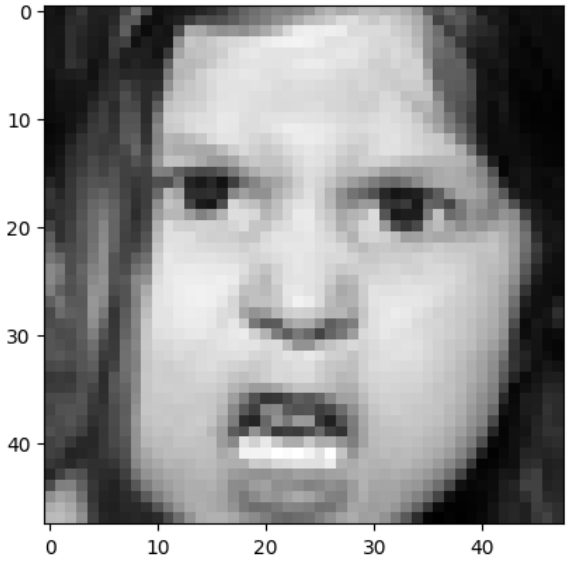
In [ ]:
```
img, label = test_ds[0]
plt.imshow(img[0], interpolation='nearest', cmap='gray')
img = img.to(device)
print('Label:', Labels[label], ', Predicted:', predict_image(img, model))
```

```
Label: Angry , Predicted: Angry
```



In [ ]:
```
img, label = test_ds[110]
plt.imshow(img[0], interpolation='nearest', cmap='gray')
img = img.to(device)
print('Label:', Labels[label], ', Predicted:', predict_image(img, model))
```

```
Label: Angry , Predicted: Fear
```



FLOPs

In deep learning, FLOPs (Floating Point Operations) quantify the total number of arithmetic operations—such as additions, multiplications, and divisions—that a model performs during a single forward pass (i.e., when making a prediction). This metric serves as an indicator of a model's computational complexity. When discussing large-scale models, FLOPs are often expressed in GFLOPs (Giga Floating Point Operations), where 1 GFLOP equals one billion operations. This unit helps in comparing the computational demands of different models.

In [ ]:
```
# we use fvcore to calculate the FLOPs
!pip install fvcore
```

```
Requirement already satisfied: fvcore in /usr/local/lib/python3.11/dist-packages (0.1.5.post20221221)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from fvcore) (2.0.2)
Requirement already satisfied: yacs>=0.1.6 in /usr/local/lib/python3.11/dist-packages (from fvcore) (0.1.8)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-packages (from fvcore) (6.0.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from fvcore) (4.67.1)
Requirement already satisfied: termcolor>=1.1 in /usr/local/lib/python3.11/dist-packages (from fvcore) (3.1.0)
Requirement already satisfied: Pillow in /usr/local/lib/python3.11/dist-packages (from fvcore) (11.2.1)
Requirement already satisfied: tabulate in /usr/local/lib/python3.11/dist-packages (from fvcore) (0.9.0)
Requirement already satisfied: iopath>=0.1.7 in /usr/local/lib/python3.11/dist-packages (from fvcore) (0.1.10)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.11/dist-packages (from iopath>=0.1.7-
>fvcore) (4.13.2)
Requirement already satisfied: portalocker in /usr/local/lib/python3.11/dist-packages (from iopath>=0.1.7->fvcor
e) (3.1.1)
```

In [ ]:
```
from fvcore.nn import FlopCountAnalysis
input = torch.randn(1, 1, 48, 48) # The input size should be the same as the size that you put into your model
```

```
#Get the network and its FLOPs
num_classes = 7
model = expression(num_classes)
flops = FlopCountAnalysis(model, input)
print(f"FLOPs: {flops.total()/1e9:.5f} GFLOPs")
```

```
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::max_pool2d encountered 3 time(s)
FLOPs: 0.32751 GFLOPs
```

In [ ]:
```
# Baseline efficiency
acc = result['val_acc']
efficiency = acc / (flops.total()/1e9)
print(f"Baseline Efficiency: {efficiency}")
```

```
Baseline Efficiency: 1.5167767379497141
```

Experiment with ResEmoteNet

In [ ]:
```
# ——— SE and Residual blocks ———
class SEBlock(nn.Module):
    def __init__(self, in_channels, reduction=16):
        super().__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(in_channels, in_channels // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(in_channels // reduction, in_channels, bias=False),
            nn.Sigmoid()
        )
    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)


class ResidualBlock(nn.Module):
    def __init__(self, in_ch, out_ch, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1   = nn.BatchNorm2d(out_ch)
        self.conv2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2   = nn.BatchNorm2d(out_ch)

        # if shape changes, match it with a 1×1 conv
        self.shortcut = nn.Sequential()
        if stride != 1 or in_ch != out_ch:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_ch, out_ch, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_ch)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out = out + self.shortcut(x)
        return F.relu(out)


# ——— Integrated model ———
class ResExpressionModel(expression_model):
    def __init__(self, num_classes, in_channels=1):
        super().__init__()
        # initial conv layers
        self.conv1 = nn.Conv2d(in_channels, 64, kernel_size=3, padding=1, bias=False)
        self.bn1   = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1, bias=False)
        self.bn2   = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1, bias=False)
        self.bn3   = nn.BatchNorm2d(256)

        self.pool  = nn.MaxPool2d(2, 2)
        self.dropout1 = nn.Dropout(0.2)

        # squeeze-and-excitation
        self.se = SEBlock(256)

        # residual layers
        self.res1 = ResidualBlock(256, 512, stride=2)
        self.res2 = ResidualBlock(512, 1024, stride=2)
        self.res3 = ResidualBlock(1024, 2048, stride=2)

        # classifier head
        self.global_pool = nn.AdaptiveAvgPool2d((1,1))
```

```
        self.fc1 = nn.Linear(2048, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 256)
        self.fc4 = nn.Linear(256, num_classes)
        self.dropout2 = nn.Dropout(0.5)

    def forward(self, x):
        # Stem
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.pool(x)
        x = self.dropout1(x)

        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool(x)
        x = self.dropout1(x)

        x = F.relu(self.bn3(self.conv3(x)))
        x = self.pool(x)

        # SE
        x = self.se(x)

        # Residual stacks
        x = self.res1(x)
        x = self.res2(x)
        x = self.res3(x)

        # Head
        x = self.global_pool(x)              # [B, 2048, 1, 1]
        x = x.view(x.size(0), -1)            # [B, 2048]
        x = F.relu(self.fc1(x))
        x = self.dropout2(x)
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)
        x = F.relu(self.fc3(x))
        x = self.dropout2(x)
        x = self.fc4(x)
        return x
```

In [ ]:

```
model_ren = to_device(ResExpressionModel(num_classes = 7, in_channels=1), device)
model_ren.apply(init_weights)
print(count_params(model_ren))
print(model_ren)
```

```
80226247
ResExpressionModel(
  (conv1): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout1): Dropout(p=0.2, inplace=False)
  (se): SEBlock(
    (avg_pool): AdaptiveAvgPool2d(output_size=1)
    (fc): Sequential(
      (0): Linear(in_features=256, out_features=16, bias=False)
      (1): ReLU(inplace=True)
      (2): Linear(in_features=16, out_features=256, bias=False)
      (3): Sigmoid()
    )
  )
  (res1): ResidualBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (res2): ResidualBlock(
    (conv1): Conv2d(512, 1024, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (res3): ResidualBlock(
    (conv1): Conv2d(1024, 2048, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(2048, 2048, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential(
      (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (global_pool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc1): Linear(in_features=2048, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=256, bias=True)
  (fc4): Linear(in_features=256, out_features=7, bias=True)
  (dropout2): Dropout(p=0.5, inplace=False)
)
```

In [ ]:

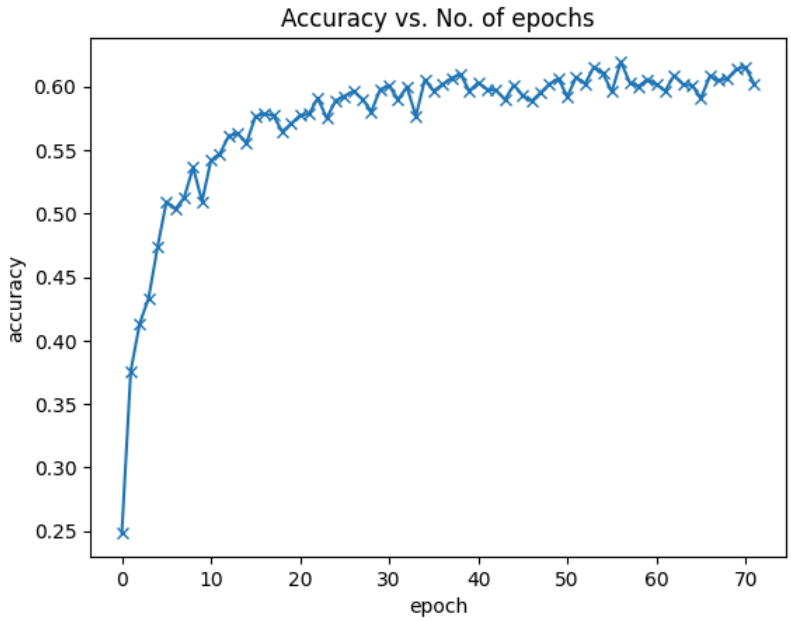
```python
num_epochs = 80
lr = 0.001
patience = 15

best_val_acc = 0.0
epochs_no_improve = 0

history_ren = []
optimizer = torch.optim.AdamW(model_ren.parameters(), lr, weight_decay=1e-4)

for epoch in range(num_epochs):
    # Training Phase
    model_ren.train()
    train_losses = []
    for batch in train_dl:
        loss = model_ren.training_step(batch)
        train_losses.append(loss)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    # Validation phase
    result = evaluate(model_ren, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_ren.epoch_end(epoch, result)
    history_ren.append(result)

    # Early-stopping + checkpointing
    val_acc = result['val_acc']
```

```
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        epochs_no_improve = 0
        # save current best model
        torch.save(model_ren.state_dict(), 'model_ren.pth')
        print(f"→ New best val_acc={val_acc:.4f}, checkpoint saved.")
    else:
        epochs_no_improve += 1
        if epochs_no_improve >= patience:
            print(f"Early stopping at epoch {epoch+1} (no val_acc improvement for {patience} epochs).")
            break
```

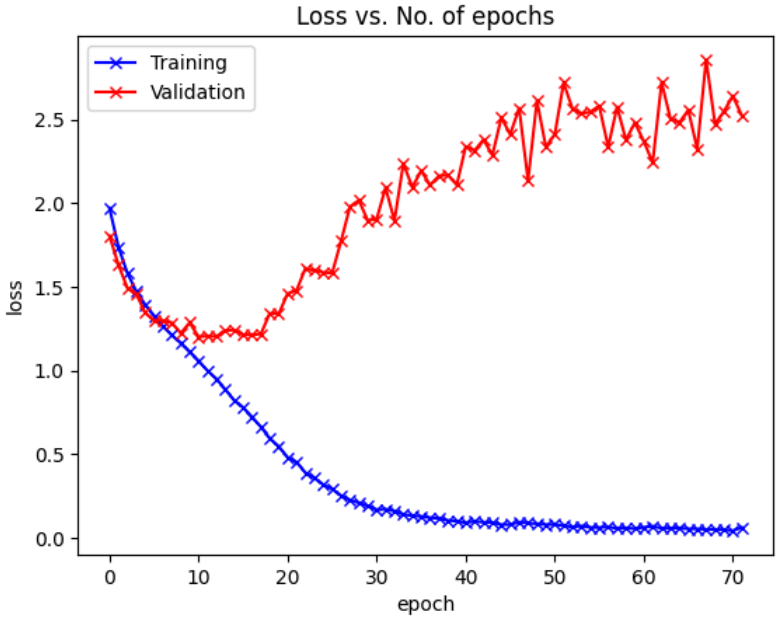
```
Epoch[0], val_loss: 1.8042, val_acc: 0.2486
→ New best val_acc=0.2486, checkpoint saved.
Epoch[1], val_loss: 1.6313, val_acc: 0.3760
→ New best val_acc=0.3760, checkpoint saved.
Epoch[2], val_loss: 1.4940, val_acc: 0.4134
→ New best val_acc=0.4134, checkpoint saved.
Epoch[3], val_loss: 1.4582, val_acc: 0.4332
→ New best val_acc=0.4332, checkpoint saved.
Epoch[4], val_loss: 1.3458, val_acc: 0.4733
→ New best val_acc=0.4733, checkpoint saved.
Epoch[5], val_loss: 1.2977, val_acc: 0.5093
→ New best val_acc=0.5093, checkpoint saved.
Epoch[6], val_loss: 1.2980, val_acc: 0.5034
Epoch[7], val_loss: 1.2860, val_acc: 0.5127
→ New best val_acc=0.5127, checkpoint saved.
Epoch[8], val_loss: 1.2231, val_acc: 0.5364
→ New best val_acc=0.5364, checkpoint saved.
Epoch[9], val_loss: 1.2867, val_acc: 0.5095
Epoch[10], val_loss: 1.2012, val_acc: 0.5418
→ New best val_acc=0.5418, checkpoint saved.
Epoch[11], val_loss: 1.2071, val_acc: 0.5471
→ New best val_acc=0.5471, checkpoint saved.
Epoch[12], val_loss: 1.2030, val_acc: 0.5612
→ New best val_acc=0.5612, checkpoint saved.
Epoch[13], val_loss: 1.2389, val_acc: 0.5628
→ New best val_acc=0.5628, checkpoint saved.
Epoch[14], val_loss: 1.2427, val_acc: 0.5557
Epoch[15], val_loss: 1.2159, val_acc: 0.5765
→ New best val_acc=0.5765, checkpoint saved.
Epoch[16], val_loss: 1.2127, val_acc: 0.5787
→ New best val_acc=0.5787, checkpoint saved.
Epoch[17], val_loss: 1.2192, val_acc: 0.5773
Epoch[18], val_loss: 1.3427, val_acc: 0.5646
Epoch[19], val_loss: 1.3386, val_acc: 0.5707
Epoch[20], val_loss: 1.4604, val_acc: 0.5773
Epoch[21], val_loss: 1.4777, val_acc: 0.5784
Epoch[22], val_loss: 1.6118, val_acc: 0.5908
→ New best val_acc=0.5908, checkpoint saved.
Epoch[23], val_loss: 1.5964, val_acc: 0.5751
Epoch[24], val_loss: 1.5872, val_acc: 0.5890
Epoch[25], val_loss: 1.5821, val_acc: 0.5924
→ New best val_acc=0.5924, checkpoint saved.
Epoch[26], val_loss: 1.7725, val_acc: 0.5965
→ New best val_acc=0.5965, checkpoint saved.
Epoch[27], val_loss: 1.9789, val_acc: 0.5902
Epoch[28], val_loss: 2.0166, val_acc: 0.5803
Epoch[29], val_loss: 1.8969, val_acc: 0.5977
→ New best val_acc=0.5977, checkpoint saved.
Epoch[30], val_loss: 1.9049, val_acc: 0.6007
→ New best val_acc=0.6007, checkpoint saved.
Epoch[31], val_loss: 2.0919, val_acc: 0.5897
Epoch[32], val_loss: 1.8955, val_acc: 0.5999
Epoch[33], val_loss: 2.2369, val_acc: 0.5767
Epoch[34], val_loss: 2.0957, val_acc: 0.6052
→ New best val_acc=0.6052, checkpoint saved.
Epoch[35], val_loss: 2.1931, val_acc: 0.5967
Epoch[36], val_loss: 2.1079, val_acc: 0.6017
Epoch[37], val_loss: 2.1634, val_acc: 0.6066
→ New best val_acc=0.6066, checkpoint saved.
Epoch[38], val_loss: 2.1671, val_acc: 0.6099
→ New best val_acc=0.6099, checkpoint saved.
Epoch[39], val_loss: 2.1091, val_acc: 0.5965
Epoch[40], val_loss: 2.3388, val_acc: 0.6030
Epoch[41], val_loss: 2.3145, val_acc: 0.5978
Epoch[42], val_loss: 2.3794, val_acc: 0.5970
Epoch[43], val_loss: 2.2872, val_acc: 0.5897
Epoch[44], val_loss: 2.5109, val_acc: 0.6013
Epoch[45], val_loss: 2.4108, val_acc: 0.5929
Epoch[46], val_loss: 2.5676, val_acc: 0.5885
Epoch[47], val_loss: 2.1355, val_acc: 0.5953
Epoch[48], val_loss: 2.6112, val_acc: 0.6020
Epoch[49], val_loss: 2.3364, val_acc: 0.6063
Epoch[50], val_loss: 2.4090, val_acc: 0.5919
Epoch[51], val_loss: 2.7187, val_acc: 0.6075
Epoch[52], val_loss: 2.5612, val_acc: 0.6023
Epoch[53], val_loss: 2.5348, val_acc: 0.6156
→ New best val_acc=0.6156, checkpoint saved.
Epoch[54], val_loss: 2.5452, val_acc: 0.6104
Epoch[55], val_loss: 2.5790, val_acc: 0.5959
Epoch[56], val_loss: 2.3359, val_acc: 0.6197
→ New best val_acc=0.6197, checkpoint saved.
Epoch[57], val_loss: 2.5744, val_acc: 0.6033
Epoch[58], val_loss: 2.3775, val_acc: 0.6002
Epoch[59], val_loss: 2.4812, val_acc: 0.6057
Epoch[60], val_loss: 2.3688, val_acc: 0.6015
Epoch[61], val_loss: 2.2422, val_acc: 0.5967
```

```
Epoch[62], val_loss: 2.7207, val_acc: 0.6089
Epoch[63], val_loss: 2.5022, val_acc: 0.6016
Epoch[64], val_loss: 2.4819, val_acc: 0.6010
Epoch[65], val_loss: 2.5558, val_acc: 0.5908
Epoch[66], val_loss: 2.3236, val_acc: 0.6080
Epoch[67], val_loss: 2.8571, val_acc: 0.6047
Epoch[68], val_loss: 2.4713, val_acc: 0.6065
Epoch[69], val_loss: 2.5482, val_acc: 0.6139
Epoch[70], val_loss: 2.6396, val_acc: 0.6153
Epoch[71], val_loss: 2.5239, val_acc: 0.6018
Early stopping at epoch 72 (no val_acc improvement for 15 epochs).
```

In [ ]:
```
plot_accuracies(history_ren)
```



In [ ]:
```
plot_losses(history_ren)
```



In [ ]:
```
from fvcore.nn import FlopCountAnalysis
model_ren.load_state_dict(torch.load('model_ren.pth'))
result_ren = evaluate(model_ren, test_dl)
print("Test result:", result_ren)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops = FlopCountAnalysis(model_ren.eval(), dummy_input)
gflops = flops.total() / 1e9
acc = result_ren['val_acc']
efficiency = acc / gflops
print(f"ResEmoteNet GFLOPs: {gflops}")
print(f"ResEmoteNet Accuracy: {acc}")
print(f"ResEmoteNet Efficiency: {efficiency}")
```

```
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::max_pool2d encountered 3 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::sigmoid encountered 1 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::expand_as encountered 1 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::mul encountered 1 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::add encountered 3 time(s)
Test result: {'val_loss': 2.3503522872924805, 'val_acc': 0.6136530041694641}
ResEmoteNet GFLOPs: 0.240086784
ResEmoteNet Accuracy: 0.6136530041694641
ResEmoteNet Efficiency: 2.555963281050339
```

VGG 19 (modified)

In [ ]:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

# 1) Configuration for VGG-19 with BatchNorm
_cfg_vgg19_bn = [
    64, 64, 'M',
    128, 128, 'M',
    256, 256, 256, 256, 'M',
    512, 512, 512, 512, 'M',
    512, 512, 512, 512, 'M',
]

def make_layers(cfg, in_channels=1):
    layers = []
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            layers += [
                nn.Conv2d(in_channels, v, kernel_size=3, padding=1, bias=False),
                nn.BatchNorm2d(v),
                nn.ReLU(inplace=True)
            ]
            in_channels = v
    return nn.Sequential(*layers)

# 2) VGG-19 model
class VGG19ExpressionModel(expression_model):
    def __init__(self, num_classes=7, in_channels=1):
        super().__init__()
        # feature extractor (5 × conv-blocks + pool)
        self.features = make_layers(_cfg_vgg19_bn, in_channels)
        # reduce whatever spatial size remains to 1×1
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        # classifier head (512→4096→4096→num_classes)
        self.classifier = nn.Sequential(
            nn.Linear(512, 4096),
            nn.ReLU(True),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(0.5),
            nn.Linear(4096, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)           # [B, 512, 1, 1]
        x = x.view(x.size(0), -1)     # [B, 512]
        x = self.classifier(x)        # [B, num_classes]
        return x

# 3) Instantiate and sanity-check
model_vgg = to_device(VGG19ExpressionModel(num_classes=7, in_channels=1), device)
model_vgg.apply(init_weights)
print(model_vgg)
print(f"Total params: {count_params(model_vgg)}")
```

```
VGG19ExpressionModel(
  (features): Sequential(
    (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (24): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (25): ReLU(inplace=True)
    (26): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (27): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (32): ReLU(inplace=True)
    (33): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (34): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (35): ReLU(inplace=True)
    (36): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (37): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (38): ReLU(inplace=True)
    (39): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (42): ReLU(inplace=True)
    (43): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (44): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (45): ReLU(inplace=True)
    (46): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (47): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (48): ReLU(inplace=True)
    (49): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (50): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (51): ReLU(inplace=True)
    (52): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (classifier): Sequential(
    (0): Linear(in_features=512, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=7, bias=True)
  )
)
Total params: 38939975
```

In [ ]:

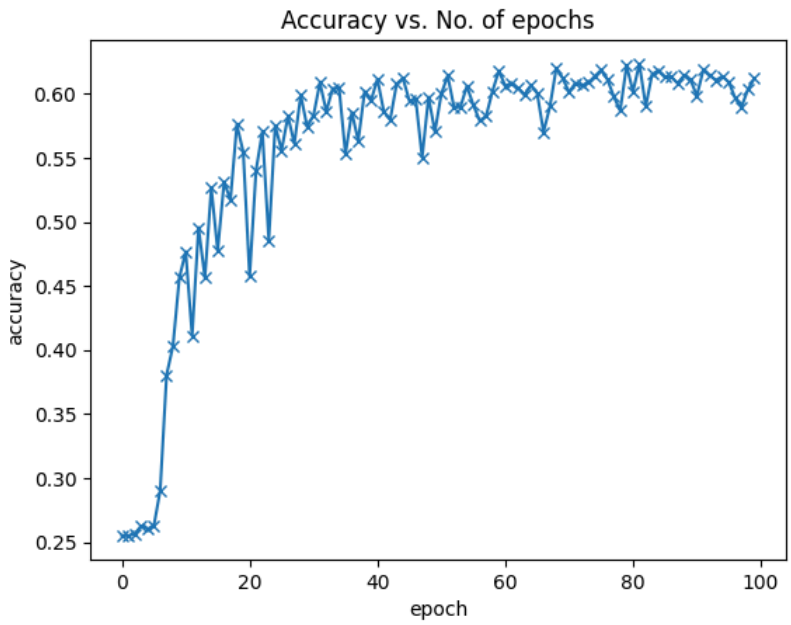
```python
num_epochs = 100
lr = 0.001
patience = 20

best_val_acc = 0.0
epochs_no_improve = 0

history_vgg = []
optimizer = torch.optim.AdamW(model_vgg.parameters(), lr, weight_decay=1e-4)

for epoch in range(num_epochs):
    # Training Phase
    model_vgg.train()
    train_losses = []
    for batch in train_dl:
        loss = model_vgg.training_step(batch)
        train_losses.append(loss)
```

```
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    # Validation phase
    result = evaluate(model_vgg, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_vgg.epoch_end(epoch, result)
    history_vgg.append(result)

    # Early-stopping + checkpointing
    val_acc = result['val_acc']
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        epochs_no_improve = 0
        # save current best model
        torch.save(model_vgg.state_dict(), 'model_vgg.pth')
        print(f"→ New best val_acc={val_acc:.4f}, checkpoint saved.")
    else:
        epochs_no_improve += 1
        if epochs_no_improve >= patience:
            print(f"Early stopping at epoch {epoch+1} (no val_acc improvement for {patience} epochs).")
            break
```

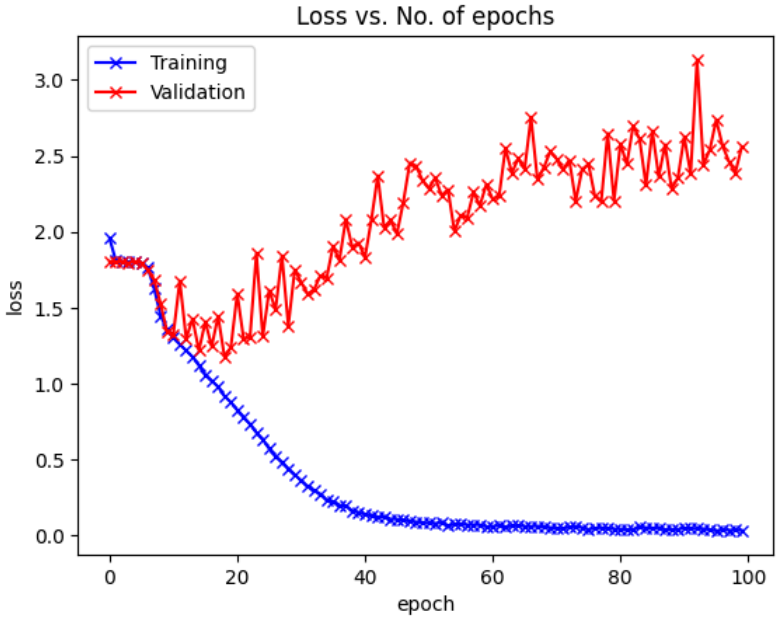
```
Epoch[0], val_loss: 1.8066, val_acc: 0.2553
→ New best val_acc=0.2553, checkpoint saved.
Epoch[1], val_loss: 1.8019, val_acc: 0.2551
Epoch[2], val_loss: 1.8044, val_acc: 0.2561
→ New best val_acc=0.2561, checkpoint saved.
Epoch[3], val_loss: 1.7961, val_acc: 0.2626
→ New best val_acc=0.2626, checkpoint saved.
Epoch[4], val_loss: 1.8048, val_acc: 0.2604
Epoch[5], val_loss: 1.7958, val_acc: 0.2624
Epoch[6], val_loss: 1.7494, val_acc: 0.2902
→ New best val_acc=0.2902, checkpoint saved.
Epoch[7], val_loss: 1.6813, val_acc: 0.3805
→ New best val_acc=0.3805, checkpoint saved.
Epoch[8], val_loss: 1.5275, val_acc: 0.4031
→ New best val_acc=0.4031, checkpoint saved.
Epoch[9], val_loss: 1.3398, val_acc: 0.4564
→ New best val_acc=0.4564, checkpoint saved.
Epoch[10], val_loss: 1.3275, val_acc: 0.4768
→ New best val_acc=0.4768, checkpoint saved.
Epoch[11], val_loss: 1.6784, val_acc: 0.4112
Epoch[12], val_loss: 1.2981, val_acc: 0.4956
→ New best val_acc=0.4956, checkpoint saved.
Epoch[13], val_loss: 1.4267, val_acc: 0.4565
Epoch[14], val_loss: 1.2185, val_acc: 0.5266
→ New best val_acc=0.5266, checkpoint saved.
Epoch[15], val_loss: 1.4085, val_acc: 0.4775
Epoch[16], val_loss: 1.2453, val_acc: 0.5312
→ New best val_acc=0.5312, checkpoint saved.
Epoch[17], val_loss: 1.4446, val_acc: 0.5173
Epoch[18], val_loss: 1.1763, val_acc: 0.5760
→ New best val_acc=0.5760, checkpoint saved.
Epoch[19], val_loss: 1.2394, val_acc: 0.5546
Epoch[20], val_loss: 1.5927, val_acc: 0.4574
Epoch[21], val_loss: 1.2947, val_acc: 0.5402
Epoch[22], val_loss: 1.3052, val_acc: 0.5704
Epoch[23], val_loss: 1.8588, val_acc: 0.4858
Epoch[24], val_loss: 1.3109, val_acc: 0.5751
Epoch[25], val_loss: 1.6145, val_acc: 0.5560
Epoch[26], val_loss: 1.4872, val_acc: 0.5826
→ New best val_acc=0.5826, checkpoint saved.
Epoch[27], val_loss: 1.8426, val_acc: 0.5611
Epoch[28], val_loss: 1.3817, val_acc: 0.5990
→ New best val_acc=0.5990, checkpoint saved.
Epoch[29], val_loss: 1.7504, val_acc: 0.5742
Epoch[30], val_loss: 1.6686, val_acc: 0.5833
Epoch[31], val_loss: 1.5916, val_acc: 0.6087
→ New best val_acc=0.6087, checkpoint saved.
Epoch[32], val_loss: 1.6170, val_acc: 0.5864
Epoch[33], val_loss: 1.7111, val_acc: 0.6042
Epoch[34], val_loss: 1.6955, val_acc: 0.6049
Epoch[35], val_loss: 1.9068, val_acc: 0.5530
Epoch[36], val_loss: 1.8115, val_acc: 0.5856
Epoch[37], val_loss: 2.0836, val_acc: 0.5633
Epoch[38], val_loss: 1.8996, val_acc: 0.6020
Epoch[39], val_loss: 1.9281, val_acc: 0.5948
Epoch[40], val_loss: 1.8310, val_acc: 0.6115
→ New best val_acc=0.6115, checkpoint saved.
Epoch[41], val_loss: 2.0768, val_acc: 0.5863
Epoch[42], val_loss: 2.3698, val_acc: 0.5798
Epoch[43], val_loss: 2.0257, val_acc: 0.6077
Epoch[44], val_loss: 2.0773, val_acc: 0.6122
→ New best val_acc=0.6122, checkpoint saved.
Epoch[45], val_loss: 1.9867, val_acc: 0.5946
Epoch[46], val_loss: 2.1937, val_acc: 0.5956
Epoch[47], val_loss: 2.4505, val_acc: 0.5502
Epoch[48], val_loss: 2.4349, val_acc: 0.5975
Epoch[49], val_loss: 2.3388, val_acc: 0.5704
Epoch[50], val_loss: 2.2847, val_acc: 0.5999
Epoch[51], val_loss: 2.3561, val_acc: 0.6141
→ New best val_acc=0.6141, checkpoint saved.
Epoch[52], val_loss: 2.2395, val_acc: 0.5893
Epoch[53], val_loss: 2.2786, val_acc: 0.5899
Epoch[54], val_loss: 2.0055, val_acc: 0.6060
Epoch[55], val_loss: 2.1076, val_acc: 0.5914
Epoch[56], val_loss: 2.0932, val_acc: 0.5791
Epoch[57], val_loss: 2.2615, val_acc: 0.5824
Epoch[58], val_loss: 2.1729, val_acc: 0.6018
Epoch[59], val_loss: 2.3126, val_acc: 0.6181
→ New best val_acc=0.6181, checkpoint saved.
Epoch[60], val_loss: 2.2149, val_acc: 0.6063
Epoch[61], val_loss: 2.2346, val_acc: 0.6082
Epoch[62], val_loss: 2.5507, val_acc: 0.6043
Epoch[63], val_loss: 2.3879, val_acc: 0.5990
Epoch[64], val_loss: 2.4824, val_acc: 0.6065
Epoch[65], val_loss: 2.4131, val_acc: 0.6006
Epoch[66], val_loss: 2.7536, val_acc: 0.5701
```

```
Epoch[67], val_loss: 2.3463, val_acc: 0.5901
Epoch[68], val_loss: 2.4217, val_acc: 0.6203
→ New best val_acc=0.6203, checkpoint saved.
Epoch[69], val_loss: 2.5285, val_acc: 0.6121
Epoch[70], val_loss: 2.4808, val_acc: 0.6013
Epoch[71], val_loss: 2.4092, val_acc: 0.6086
Epoch[72], val_loss: 2.4723, val_acc: 0.6066
Epoch[73], val_loss: 2.2032, val_acc: 0.6087
Epoch[74], val_loss: 2.4101, val_acc: 0.6135
Epoch[75], val_loss: 2.4459, val_acc: 0.6185
Epoch[76], val_loss: 2.2351, val_acc: 0.6113
Epoch[77], val_loss: 2.2000, val_acc: 0.5978
Epoch[78], val_loss: 2.6484, val_acc: 0.5869
Epoch[79], val_loss: 2.2044, val_acc: 0.6222
→ New best val_acc=0.6222, checkpoint saved.
Epoch[80], val_loss: 2.5805, val_acc: 0.6018
Epoch[81], val_loss: 2.4459, val_acc: 0.6235
→ New best val_acc=0.6235, checkpoint saved.
Epoch[82], val_loss: 2.6983, val_acc: 0.5901
Epoch[83], val_loss: 2.6118, val_acc: 0.6156
Epoch[84], val_loss: 2.3073, val_acc: 0.6177
Epoch[85], val_loss: 2.6646, val_acc: 0.6132
Epoch[86], val_loss: 2.3642, val_acc: 0.6131
Epoch[87], val_loss: 2.5742, val_acc: 0.6081
Epoch[88], val_loss: 2.2847, val_acc: 0.6144
Epoch[89], val_loss: 2.3548, val_acc: 0.6110
Epoch[90], val_loss: 2.6260, val_acc: 0.5984
Epoch[91], val_loss: 2.3821, val_acc: 0.6186
Epoch[92], val_loss: 3.1338, val_acc: 0.6142
Epoch[93], val_loss: 2.4432, val_acc: 0.6108
Epoch[94], val_loss: 2.5394, val_acc: 0.6135
Epoch[95], val_loss: 2.7380, val_acc: 0.6092
Epoch[96], val_loss: 2.5656, val_acc: 0.5966
Epoch[97], val_loss: 2.4627, val_acc: 0.5899
Epoch[98], val_loss: 2.3895, val_acc: 0.6036
Epoch[99], val_loss: 2.5608, val_acc: 0.6119
```

In [ ]:
```
plot_accuracies(history_vgg)
```



In [ ]:
```
plot_losses(history_vgg)
```



In [ ]:
```python
from fvcore.nn import FlopCountAnalysis
# ———————— final test ————————————————————————————————————
model_vgg.load_state_dict(torch.load('model_vgg.pth'))

# 6) Final evaluation on the test set
result_vgg = evaluate(model_vgg, test_dl)
print("Test result:", result_vgg)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops = FlopCountAnalysis(model_vgg.eval(), dummy_input)
gflops = flops.total() / 1e9
acc = result_vgg['val_acc']
efficiency = acc / gflops
print(f"VGG GFLOPs: {gflops}")
print(f"VGG Accuracy: {acc}")
print(f"VGG Efficiency: {efficiency}")
```

```
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::max_pool2d encountered 5 time(s)
Test result: {'val_loss': 2.4295873641967773, 'val_acc': 0.6200584769248962}
VGG GFLOPs: 0.913408512
VGG Accuracy: 0.6200584769248962
VGG Efficiency: 0.6788402656410719
```

EfficientNet b1

In [ ]:
```python
from torchvision.models import efficientnet_v2_s, efficientnet_b1
from torch import nn
import torch.nn.functional as F


class EfficientNet(expression_model):
    def __init__(self, number_of_class: int, in_channel: int = 1):
        super().__init__()

        # ---- backbone (no weights) ----------------------------------------
        # self.model = efficientnet_v2_s(weights=None)
        self.model = efficientnet_b1(weights=None)

        # ---- replace stem for 1-channel input -----------------------------
        # features[0] = ConvNormActivation(conv, bn, silu)
        old_conv = self.model.features[0][0]          # nn.Conv2d
        self.model.features[0][0] = nn.Conv2d(
            in_channels  = in_channel,
            out_channels  = old_conv.out_channels,    # 24
            kernel_size   = old_conv.kernel_size,
            stride        = old_conv.stride,
            padding       = old_conv.padding,
            bias=False
        )

        # ---- replace classifier head --------------------------------------
        in_feat = self.model.classifier[1].in_features  # 1280 for v2-s
        self.model.classifier[1] = nn.Linear(in_feat, number_of_class)

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

```
    def forward(self, x):
        return self.model(x)        # returns logits (batch, num_classes)

model_eff = to_device(EfficientNet(7,1),device)
model_eff.apply(init_weights)
print(count_params(model_eff))
print(model_eff)
```

```
6521575
EfficientNet(
  (model): EfficientNet(
    (features): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
              (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(32, 8, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(8, 32, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (2): Conv2dNormActivation(
              (0): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.0, mode=row)
        )
        (1): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=16, bias=False)
              (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(16, 4, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(4, 16, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (2): Conv2dNormActivation(
              (0): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.008695652173913044, mode=row)
        )
      )
      (2): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(16, 96, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=96, bias=False)
              (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(96, 4, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(4, 96, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(96, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.017391304347826087, mode=row)
        )
        (1): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
```

```
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(144, 144, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=144, bias=False)
        (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(144, 6, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(6, 144, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(144, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.026086956521739136, mode=row)
  )
  (2): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(144, 144, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=144, bias=False)
        (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(144, 6, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(6, 144, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(144, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.034782608695652174, mode=row)
  )
)
(3): Sequential(
  (0): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(144, 144, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), groups=144, bias=False)
        (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(144, 6, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(6, 144, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(144, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.043478260869565216, mode=row)
  )
  (1): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(240, 240, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=240, bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
```

```
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(240, 10, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(10, 240, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(240, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.05217391304347827, mode=row)
  )
  (2): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(240, 240, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=240, bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(240, 10, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(10, 240, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(240, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.06086956521739131, mode=row)
  )
)
(4): Sequential(
  (0): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(240, 240, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=240, bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(240, 10, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(10, 240, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(240, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.06956521739130435, mode=row)
  )
  (1): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=480, bias=False)
        (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
```

```
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(480, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (stochastic_depth): StochasticDepth(p=0.0782608695652174, mode=row)
    )
    (2): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=480, bias=False)
          (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
          (activation): SiLU(inplace=True)
          (scale_activation): Sigmoid()
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(480, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (stochastic_depth): StochasticDepth(p=0.08695652173913043, mode=row)
    )
    (3): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=480, bias=False)
          (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
          (activation): SiLU(inplace=True)
          (scale_activation): Sigmoid()
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(480, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (stochastic_depth): StochasticDepth(p=0.09565217391304348, mode=row)
    )
  )
  (5): Sequential(
    (0): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(480, 480, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=480, bias=False)
          (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
          (activation): SiLU(inplace=True)
          (scale_activation): Sigmoid()
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(480, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
```

```
      (stochastic_depth): StochasticDepth(p=0.10434782608695654, mode=row)
    )
    (1): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=672, bias=False)
          (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
          (activation): SiLU(inplace=True)
          (scale_activation): Sigmoid()
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (stochastic_depth): StochasticDepth(p=0.11304347826086956, mode=row)
    )
    (2): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=672, bias=False)
          (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
          (activation): SiLU(inplace=True)
          (scale_activation): Sigmoid()
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (stochastic_depth): StochasticDepth(p=0.12173913043478261, mode=row)
    )
    (3): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=672, bias=False)
          (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
          (activation): SiLU(inplace=True)
          (scale_activation): Sigmoid()
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (stochastic_depth): StochasticDepth(p=0.13043478260869565, mode=row)
    )
  )
  (6): Sequential(
    (0): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), groups=672, bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(672, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.1391304347826087, mode=row)
      )
      (1): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=1152, bias=Fals
e)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.14782608695652175, mode=row)
      )
      (2): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=1152, bias=Fals
e)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.1565217391304348, mode=row)
      )
      (3): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=1152, bias=Fals
e)
```

```
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.16521739130434784, mode=row)
      )
      (4): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=1152, bias=Fals
e)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.17391304347826086, mode=row)
      )
    )
    (7): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1152, 1152, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1152, bias=Fals
e)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1152, 320, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.1826086956521739, mode=row)
      )
      (1): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(320, 1920, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1920, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1920, 1920, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1920, bias=Fals
e)
            (1): BatchNorm2d(1920, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
```

```
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1920, 80, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(80, 1920, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1920, 320, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.19130434782608696, mode=row)
        )
      )
      (8): Conv2dNormActivation(
        (0): Conv2d(320, 1280, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(1280, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=1)
    (classifier): Sequential(
      (0): Dropout(p=0.2, inplace=True)
      (1): Linear(in_features=1280, out_features=7, bias=True)
    )
  )
)
```

In [ ]:

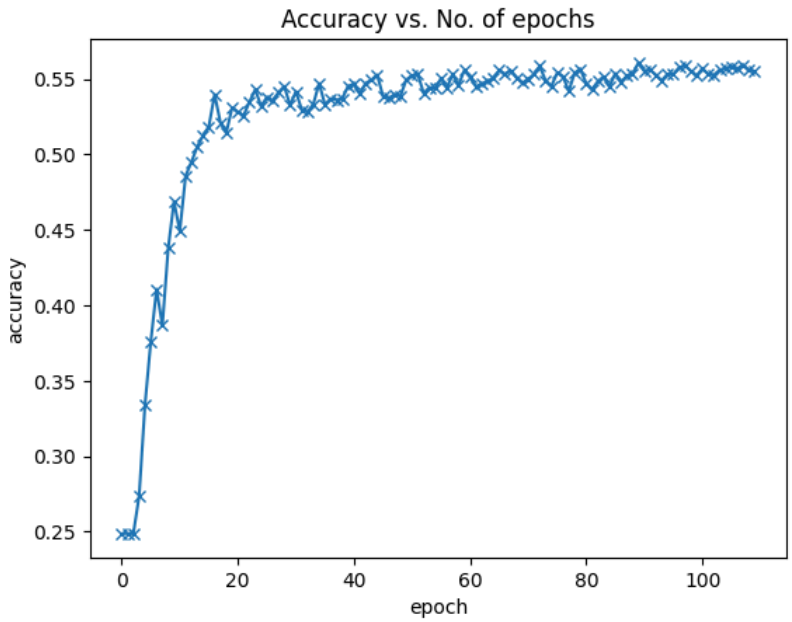
```python
# ──────────────────────────────────────────────────────────────
#  EfficientNet-B1  →  optimised training loop (FER-2013)
# ──────────────────────────────────────────────────────────────
import math, torch
from torch.cuda.amp import autocast, GradScaler

# ❶  Hyper-parameters (paper-style defaults + FER tweaks)
epochs_eff        = 120
batch_size_eff    = 128          # cut to 64 if you hit OOM
base_lr_eff       = 3e-4         # LR at start of cosine
max_lr_eff        = 3e-3         # LR peak for One-Cycle
weight_decay_eff  = 0.05
label_smooth_eff  = 0.2
grad_clip_norm    = 1.0
patience_eff      = 20

# ❷  Optimiser — parameter-wise weight-decay split
decay, no_decay = [], []
for n, p in model_eff.named_parameters():
    if not p.requires_grad: continue
    (no_decay if p.dim()==1 or n.endswith(".bias") else decay).append(p)

optimizer_eff = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_eff},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_eff, betas=(0.9, 0.9995)
)

# ❸  One-Cycle scheduler (warms-up to max_lr, cools on cosine)
sched_eff = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_eff,
    max_lr          = max_lr_eff,
    epochs          = epochs_eff,
    steps_per_epoch = len(train_dl),
    pct_start       = 0.1,
    final_div_factor= 1e4
)

# ❹  Mixed-precision scaler
scaler = GradScaler()

# ❺  Training loop with AMP, gradient-clip, early-stop, checkpoint
best_val_acc_eff   = 0.0
epochs_no_imp_eff  = 0
history_eff        = []

for epoch in range(epochs_eff):
    # ---- TRAIN ------------------------------------------------------
    model_eff.train()
    train_losses = []
    for batch in train_dl:
        with autocast():                                # mixed precision
            loss = model_eff.training_step(batch)       # CE + smooth inside
        train_losses.append(loss.detach())
        scaler.scale(loss).backward()

        # clip gradients *before* stepping
```

```
        scaler.unscale_(optimizer_eff)
        torch.nn.utils.clip_grad_norm_(model_eff.parameters(), grad_clip_norm)

        scaler.step(optimizer_eff); scaler.update()
        optimizer_eff.zero_grad()
        sched_eff.step()

    # ---- VALIDATE ------------------------------------------------------------
    result = evaluate(model_eff, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff.epoch_end(epoch, result)                  # prints nicely
    history_eff.append(result)

    # ---- EARLY-STOP / CKPT ---------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_eff:
        best_val_acc_eff, epochs_no_imp_eff = val_acc, 0
        torch.save(model_eff.state_dict(), 'model_efficientnet_best.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_eff += 1
        if epochs_no_imp_eff >= patience_eff:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_eff} epochs).")
            break
```

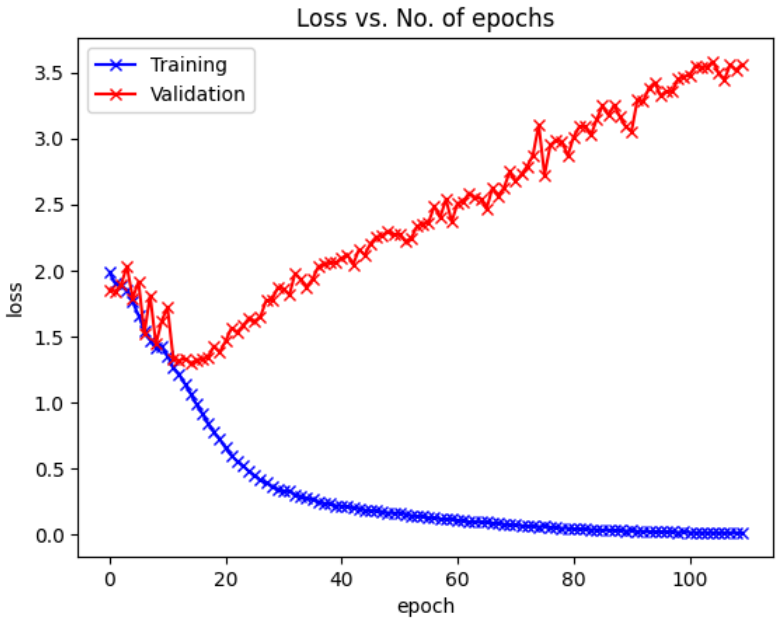
```
Epoch[0], val_loss: 1.8537, val_acc: 0.2486
→ New best val_acc = 0.2486  (ckpt saved)
Epoch[1], val_loss: 1.8405, val_acc: 0.2488
→ New best val_acc = 0.2488  (ckpt saved)
Epoch[2], val_loss: 1.8919, val_acc: 0.2486
Epoch[3], val_loss: 2.0312, val_acc: 0.2737
→ New best val_acc = 0.2737  (ckpt saved)
Epoch[4], val_loss: 1.7899, val_acc: 0.3339
→ New best val_acc = 0.3339  (ckpt saved)
Epoch[5], val_loss: 1.9154, val_acc: 0.3758
→ New best val_acc = 0.3758  (ckpt saved)
Epoch[6], val_loss: 1.5201, val_acc: 0.4107
→ New best val_acc = 0.4107  (ckpt saved)
Epoch[7], val_loss: 1.8097, val_acc: 0.3866
Epoch[8], val_loss: 1.4419, val_acc: 0.4377
→ New best val_acc = 0.4377  (ckpt saved)
Epoch[9], val_loss: 1.6188, val_acc: 0.4687
→ New best val_acc = 0.4687  (ckpt saved)
Epoch[10], val_loss: 1.7186, val_acc: 0.4493
Epoch[11], val_loss: 1.3282, val_acc: 0.4858
→ New best val_acc = 0.4858  (ckpt saved)
Epoch[12], val_loss: 1.3175, val_acc: 0.4950
→ New best val_acc = 0.4950  (ckpt saved)
Epoch[13], val_loss: 1.3315, val_acc: 0.5051
→ New best val_acc = 0.5051  (ckpt saved)
Epoch[14], val_loss: 1.2988, val_acc: 0.5125
→ New best val_acc = 0.5125  (ckpt saved)
Epoch[15], val_loss: 1.3178, val_acc: 0.5186
→ New best val_acc = 0.5186  (ckpt saved)
Epoch[16], val_loss: 1.3320, val_acc: 0.5399
→ New best val_acc = 0.5399  (ckpt saved)
Epoch[17], val_loss: 1.3394, val_acc: 0.5211
Epoch[18], val_loss: 1.4274, val_acc: 0.5142
Epoch[19], val_loss: 1.3815, val_acc: 0.5310
Epoch[20], val_loss: 1.4635, val_acc: 0.5283
Epoch[21], val_loss: 1.5608, val_acc: 0.5259
Epoch[22], val_loss: 1.5321, val_acc: 0.5349
Epoch[23], val_loss: 1.5805, val_acc: 0.5431
→ New best val_acc = 0.5431  (ckpt saved)
Epoch[24], val_loss: 1.6357, val_acc: 0.5320
Epoch[25], val_loss: 1.6171, val_acc: 0.5380
Epoch[26], val_loss: 1.6469, val_acc: 0.5355
Epoch[27], val_loss: 1.7721, val_acc: 0.5413
Epoch[28], val_loss: 1.7773, val_acc: 0.5453
→ New best val_acc = 0.5453  (ckpt saved)
Epoch[29], val_loss: 1.8682, val_acc: 0.5325
Epoch[30], val_loss: 1.8628, val_acc: 0.5418
Epoch[31], val_loss: 1.8141, val_acc: 0.5297
Epoch[32], val_loss: 1.9786, val_acc: 0.5283
Epoch[33], val_loss: 1.9292, val_acc: 0.5334
Epoch[34], val_loss: 1.8752, val_acc: 0.5473
→ New best val_acc = 0.5473  (ckpt saved)
Epoch[35], val_loss: 1.9335, val_acc: 0.5330
Epoch[36], val_loss: 2.0286, val_acc: 0.5371
Epoch[37], val_loss: 2.0500, val_acc: 0.5362
Epoch[38], val_loss: 2.0573, val_acc: 0.5371
Epoch[39], val_loss: 2.0587, val_acc: 0.5455
Epoch[40], val_loss: 2.0923, val_acc: 0.5466
Epoch[41], val_loss: 2.1201, val_acc: 0.5403
Epoch[42], val_loss: 2.0445, val_acc: 0.5473
Epoch[43], val_loss: 2.1578, val_acc: 0.5495
→ New best val_acc = 0.5495  (ckpt saved)
Epoch[44], val_loss: 2.1170, val_acc: 0.5529
→ New best val_acc = 0.5529  (ckpt saved)
Epoch[45], val_loss: 2.2040, val_acc: 0.5387
Epoch[46], val_loss: 2.2539, val_acc: 0.5375
Epoch[47], val_loss: 2.2629, val_acc: 0.5396
Epoch[48], val_loss: 2.2981, val_acc: 0.5388
Epoch[49], val_loss: 2.2697, val_acc: 0.5496
Epoch[50], val_loss: 2.2738, val_acc: 0.5524
Epoch[51], val_loss: 2.2193, val_acc: 0.5532
→ New best val_acc = 0.5532  (ckpt saved)
Epoch[52], val_loss: 2.2419, val_acc: 0.5407
Epoch[53], val_loss: 2.3341, val_acc: 0.5446
Epoch[54], val_loss: 2.3539, val_acc: 0.5439
Epoch[55], val_loss: 2.3622, val_acc: 0.5505
Epoch[56], val_loss: 2.4887, val_acc: 0.5442
Epoch[57], val_loss: 2.4001, val_acc: 0.5531
Epoch[58], val_loss: 2.5386, val_acc: 0.5459
Epoch[59], val_loss: 2.3695, val_acc: 0.5558
→ New best val_acc = 0.5558  (ckpt saved)
Epoch[60], val_loss: 2.5092, val_acc: 0.5512
Epoch[61], val_loss: 2.5220, val_acc: 0.5452
Epoch[62], val_loss: 2.5871, val_acc: 0.5467
Epoch[63], val_loss: 2.5552, val_acc: 0.5487
Epoch[64], val_loss: 2.5433, val_acc: 0.5504
```

```
Epoch[65], val_loss: 2.4634, val_acc: 0.5559
→ New best val_acc = 0.5559  (ckpt saved)
Epoch[66], val_loss: 2.6247, val_acc: 0.5535
Epoch[67], val_loss: 2.5648, val_acc: 0.5557
Epoch[68], val_loss: 2.6245, val_acc: 0.5509
Epoch[69], val_loss: 2.7482, val_acc: 0.5475
Epoch[70], val_loss: 2.6750, val_acc: 0.5500
Epoch[71], val_loss: 2.7277, val_acc: 0.5533
Epoch[72], val_loss: 2.7883, val_acc: 0.5586
→ New best val_acc = 0.5586  (ckpt saved)
Epoch[73], val_loss: 2.8720, val_acc: 0.5490
Epoch[74], val_loss: 3.1031, val_acc: 0.5454
Epoch[75], val_loss: 2.7231, val_acc: 0.5541
Epoch[76], val_loss: 2.9501, val_acc: 0.5518
Epoch[77], val_loss: 2.9842, val_acc: 0.5420
Epoch[78], val_loss: 2.9778, val_acc: 0.5541
Epoch[79], val_loss: 2.8674, val_acc: 0.5566
Epoch[80], val_loss: 3.0090, val_acc: 0.5472
Epoch[81], val_loss: 3.0884, val_acc: 0.5431
Epoch[82], val_loss: 3.0918, val_acc: 0.5485
Epoch[83], val_loss: 3.0242, val_acc: 0.5517
Epoch[84], val_loss: 3.1451, val_acc: 0.5449
Epoch[85], val_loss: 3.2463, val_acc: 0.5538
Epoch[86], val_loss: 3.1753, val_acc: 0.5480
Epoch[87], val_loss: 3.2527, val_acc: 0.5522
Epoch[88], val_loss: 3.1631, val_acc: 0.5533
Epoch[89], val_loss: 3.0878, val_acc: 0.5610
→ New best val_acc = 0.5610  (ckpt saved)
Epoch[90], val_loss: 3.0542, val_acc: 0.5553
Epoch[91], val_loss: 3.2900, val_acc: 0.5562
Epoch[92], val_loss: 3.2811, val_acc: 0.5529
Epoch[93], val_loss: 3.3810, val_acc: 0.5491
Epoch[94], val_loss: 3.4234, val_acc: 0.5534
Epoch[95], val_loss: 3.3256, val_acc: 0.5534
Epoch[96], val_loss: 3.3570, val_acc: 0.5577
Epoch[97], val_loss: 3.3528, val_acc: 0.5589
Epoch[98], val_loss: 3.4486, val_acc: 0.5556
Epoch[99], val_loss: 3.4680, val_acc: 0.5521
Epoch[100], val_loss: 3.4776, val_acc: 0.5571
Epoch[101], val_loss: 3.5492, val_acc: 0.5533
Epoch[102], val_loss: 3.5418, val_acc: 0.5528
Epoch[103], val_loss: 3.5403, val_acc: 0.5561
Epoch[104], val_loss: 3.5813, val_acc: 0.5571
Epoch[105], val_loss: 3.4982, val_acc: 0.5582
Epoch[106], val_loss: 3.4444, val_acc: 0.5568
Epoch[107], val_loss: 3.5567, val_acc: 0.5594
Epoch[108], val_loss: 3.5175, val_acc: 0.5561
Epoch[109], val_loss: 3.5597, val_acc: 0.5551
Early stopping at epoch 110 (no val_acc gain for 20 epochs).
```

In [ ]:
```
# Plotting training history
plot_accuracies(history_eff)
```



In [ ]:
```
plot_losses(history_eff)
```



In [ ]:
```
# ———————— final test ————————————————————————————————————
model_eff.load_state_dict(torch.load('model_efficientnet_best.pth'))

# 6) Final evaluation on the test set
result_eff_test = evaluate(model_eff, test_dl)
print("EfficientNet Test result:", result_eff_test)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops_eff = FlopCountAnalysis(model_eff.eval(), dummy_input)
gflops_eff = flops_eff.total() / 1e9
acc_eff = result_eff_test['val_acc']
efficiency_eff = acc_eff / gflops_eff
print(f"EfficientNet GFLOPs: {gflops_eff}")
print(f"EfficientNet Accuracy: {acc_eff}")
print(f"EfficientNet Efficiency: {efficiency_eff}")
```

```
EfficientNet Test result: {'val_loss': 2.9795453548431396, 'val_acc': 0.5616767406463623}
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::silu_ encountered 69 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::sigmoid encountered 23 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::mul encountered 23 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::add_ encountered 16 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::dropout_ encountered 1 time(s)
WARNING:fvcore.nn.jit_analysis:The following submodules of the model were never called during the trace of the g
raph. They may be unused, or they were accessed by direct calls to .forward() or via other python methods. In th
e latter case they will have zeros for statistics, though their statistics will still contribute to their parent
calling module.
model.features.1.0.stochastic_depth, model.features.1.1.stochastic_depth, model.features.2.0.stochastic_depth, m
odel.features.2.1.stochastic_depth, model.features.2.2.stochastic_depth, model.features.3.0.stochastic_depth, mo
del.features.3.1.stochastic_depth, model.features.3.2.stochastic_depth, model.features.4.0.stochastic_depth, mod
el.features.4.1.stochastic_depth, model.features.4.2.stochastic_depth, model.features.4.3.stochastic_depth, mode
l.features.5.0.stochastic_depth, model.features.5.1.stochastic_depth, model.features.5.2.stochastic_depth, mode
l.features.5.3.stochastic_depth, model.features.6.0.stochastic_depth, model.features.6.1.stochastic_depth, mode
l.features.6.2.stochastic_depth, model.features.6.3.stochastic_depth, model.features.6.4.stochastic_depth, mode
l.features.7.0.stochastic_depth, model.features.7.1.stochastic_depth
EfficientNet GFLOPs: 0.035440128
EfficientNet Accuracy: 0.5616767406463623
EfficientNet Efficiency: 15.848609255766862
```

MobileNetV3

Note: Training of this model is continued from checkpoint due to Colab corruption

In [ ]:
```
from torchvision.models import mobilenet_v3_large
from torch import nn
import torch.nn.functional as F


class MobileNetV3Large(expression_model):
    def __init__(self, number_of_class: int, in_channel: int = 1):
        super().__init__()

        # ---- backbone (no weights) ----------------------------------------
        self.model = mobilenet_v3_large(weights=None)

        # ---- replace stem for 1-channel input -----------------------------
```

```
        # features[0] = ConvNormActivation(conv, bn, hardswish)
        old_conv = self.model.features[0][0]          # nn.Conv2d
        self.model.features[0][0] = nn.Conv2d(
            in_channels   = in_channel,
            out_channels  = old_conv.out_channels,    # 16
            kernel_size   = old_conv.kernel_size,
            stride        = old_conv.stride,
            padding       = old_conv.padding,
            bias=False
        )

        # ---- replace classifier head ----------------------------------------
        in_feat = self.model.classifier[-1].in_features  # 1280 for v3-large
        self.model.classifier[-1] = nn.Linear(in_feat, number_of_class)

    # ---------------------------------------------------------------------
    def forward(self, x):
        return self.model(x)        # returns logits (batch, num_classes)

model_mobile = to_device(MobileNetV3Large(7,1), device)
model_mobile.apply(init_weights)
print(count_params(model_mobile))
print(model_mobile)
```

```
4210711
MobileNetV3Large(
  (model): MobileNetV3(
    (features): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(1, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(16, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
        (2): Hardswish()
      )
      (1): InvertedResidual(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=16, bias=False)
            (1): BatchNorm2d(16, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(16, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
          )
        )
      )
      (2): InvertedResidual(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(64, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=64, bias=False)
            (1): BatchNorm2d(64, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
          )
          (2): Conv2dNormActivation(
            (0): Conv2d(64, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(24, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
          )
        )
      )
      (3): InvertedResidual(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(24, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(72, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(72, 72, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=72, bias=False)
            (1): BatchNorm2d(72, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
          )
          (2): Conv2dNormActivation(
            (0): Conv2d(72, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(24, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
          )
        )
      )
      (4): InvertedResidual(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(24, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(72, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(72, 72, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), groups=72, bias=False)
            (1): BatchNorm2d(72, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(72, 24, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(24, 72, kernel_size=(1, 1), stride=(1, 1))
            (activation): ReLU()
            (scale_activation): Hardsigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(72, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(40, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
          )
        )
      )
      (5): InvertedResidual(
        (block): Sequential(
          (0): Conv2dNormActivation(
```

```
      (0): Conv2d(40, 120, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(120, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(120, 120, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=120, bias=False)
      (1): BatchNorm2d(120, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(120, 32, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(32, 120, kernel_size=(1, 1), stride=(1, 1))
      (activation): ReLU()
      (scale_activation): Hardsigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(120, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(40, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
    )
  )
)
(6): InvertedResidual(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(40, 120, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(120, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(120, 120, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=120, bias=False)
      (1): BatchNorm2d(120, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(120, 32, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(32, 120, kernel_size=(1, 1), stride=(1, 1))
      (activation): ReLU()
      (scale_activation): Hardsigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(120, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(40, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
    )
  )
)
(7): InvertedResidual(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(240, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(240, 240, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=240, bias=False)
      (1): BatchNorm2d(240, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (2): Conv2dNormActivation(
      (0): Conv2d(240, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(80, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
    )
  )
)
(8): InvertedResidual(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(80, 200, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(200, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(200, 200, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=200, bias=False)
      (1): BatchNorm2d(200, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (2): Conv2dNormActivation(
      (0): Conv2d(200, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(80, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
    )
  )
)
(9): InvertedResidual(
  (block): Sequential(
    (0): Conv2dNormActivation(
```

```
      (0): Conv2d(80, 184, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(184, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(184, 184, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=184, bias=False)
      (1): BatchNorm2d(184, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (2): Conv2dNormActivation(
      (0): Conv2d(184, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(80, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
    )
  )
)
(10): InvertedResidual(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(80, 184, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(184, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(184, 184, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=184, bias=False)
      (1): BatchNorm2d(184, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (2): Conv2dNormActivation(
      (0): Conv2d(184, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(80, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
    )
  )
)
(11): InvertedResidual(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(480, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=480, bias=False)
      (1): BatchNorm2d(480, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(480, 120, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(120, 480, kernel_size=(1, 1), stride=(1, 1))
      (activation): ReLU()
      (scale_activation): Hardsigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(480, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(112, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
    )
  )
)
(12): InvertedResidual(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(672, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(672, 672, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=672, bias=False)
      (1): BatchNorm2d(672, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(672, 168, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(168, 672, kernel_size=(1, 1), stride=(1, 1))
      (activation): ReLU()
      (scale_activation): Hardsigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(112, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
    )
  )
)
(13): InvertedResidual(
  (block): Sequential(
    (0): Conv2dNormActivation(
```

```
        (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(672, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
        (2): Hardswish()
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), groups=672, bias=False)
        (1): BatchNorm2d(672, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
        (2): Hardswish()
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(672, 168, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(168, 672, kernel_size=(1, 1), stride=(1, 1))
        (activation): ReLU()
        (scale_activation): Hardsigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(672, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      )
    )
  )
  (14): InvertedResidual(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
        (2): Hardswish()
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(960, 960, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=960, bias=False)
        (1): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
        (2): Hardswish()
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(960, 240, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(240, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): ReLU()
        (scale_activation): Hardsigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      )
    )
  )
  (15): InvertedResidual(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
        (2): Hardswish()
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(960, 960, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=960, bias=False)
        (1): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
        (2): Hardswish()
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(960, 240, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(240, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): ReLU()
        (scale_activation): Hardsigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      )
    )
  )
  (16): Conv2dNormActivation(
    (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (1): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
    (2): Hardswish()
  )
)
(avgpool): AdaptiveAvgPool2d(output_size=1)
(classifier): Sequential(
  (0): Linear(in_features=960, out_features=1280, bias=True)
  (1): Hardswish()
  (2): Dropout(p=0.2, inplace=True)
  (3): Linear(in_features=1280, out_features=7, bias=True)
)
```

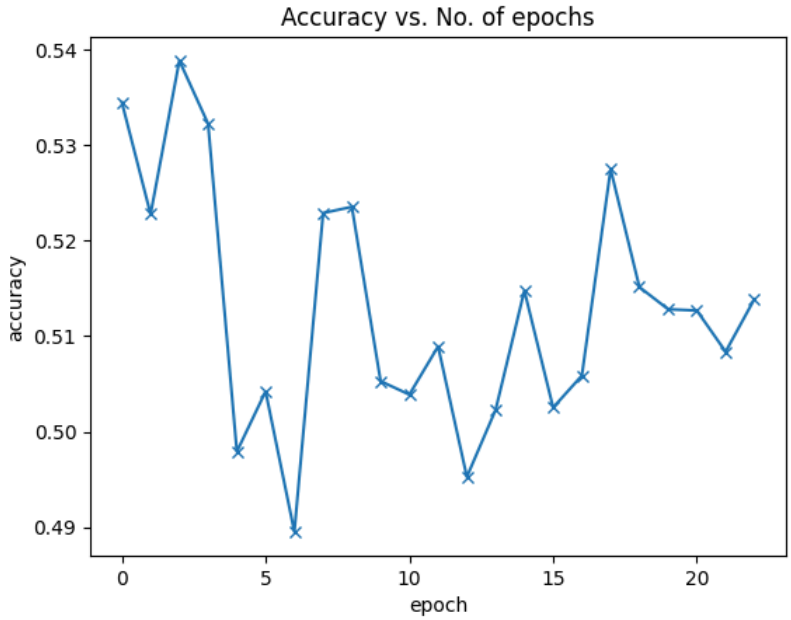
```
    )
)
```

In [ ]:

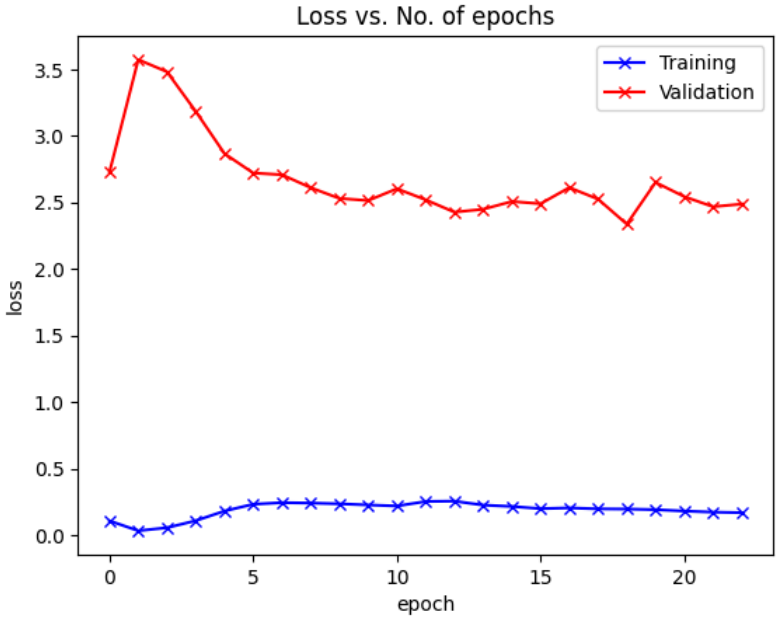
```
# ──────────────────────────────────────────────────────────────────────────
#  MobileNetV3-Large  →  AdamW-optimised training loop (FER-2013, 224×224 RGB)
# ──────────────────────────────────────────────────────────────────────────
import torch, math
from torch.cuda.amp import autocast, GradScaler

# ❶  Hyper-parameters (AdamW recipe)
epochs_mobile        = 120          # One-Cycle length
batch_size_mobile    = 192          # reduce if OOM
base_lr_mobile       = 3e-4         # LR at start of One-Cycle  ← NEW
max_lr_mobile        = 3e-3         # LR peak                   ← NEW
weight_decay_mobile  = 0.05         # ConvNeXt/ViT default      ← NEW
label_smooth_mobile  = 0.1
grad_clip_norm       = 0.5
patience_mobile      = 20

# ❷  AdamW with **param-wise weight-decay split** ----------------------------
decay, no_decay = [], []
for n, p in model_mobile.named_parameters():
    if not p.requires_grad: continue
    (no_decay if p.dim()==1 or n.endswith(".bias") else decay).append(p)

optimizer_mobile = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_mobile},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_mobile, betas=(0.9, 0.9995)          # β₂ slightly lower
)

# ❸  One-Cycle LR  (warm-up 5 %, cool-down cosine to 1e-6 × base) ------------
sched_mobile = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_mobile,
    max_lr          = max_lr_mobile,
    epochs          = epochs_mobile,
    steps_per_epoch = len(train_dl),
    pct_start       = 0.05,
    final_div_factor= 1e6                           # end LR ≈ 3e-10
)

# ❹  Mixed-precision scaler
scaler = GradScaler()

# ❺  Training loop
best_val_acc_mobile   = 0.0
epochs_no_imp_mobile  = 0
history_mobile        = []

for epoch in range(epochs_mobile):
    # ---- TRAIN ------------------------------------------------------------
    model_mobile.train()
    train_losses = []
    for batch in train_dl:
        with autocast():
            loss = model_mobile.training_step(batch)
        train_losses.append(loss.detach())

        scaler.scale(loss).backward()
        scaler.unscale_(optimizer_mobile)
        torch.nn.utils.clip_grad_norm_(model_mobile.parameters(),
                                       grad_clip_norm)

        scaler.step(optimizer_mobile);  scaler.update()
        optimizer_mobile.zero_grad()
        sched_mobile.step()

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_mobile, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_mobile.epoch_end(epoch, result)
    history_mobile.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_mobile:
        best_val_acc_mobile, epochs_no_imp_mobile = val_acc, 0
        torch.save(model_mobile.state_dict(), 'model_mobilenetv3_best.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_mobile += 1
        if epochs_no_imp_mobile >= patience_mobile:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_mobile} epochs).")
            break
```

```
Epoch[0], val_loss: 2.7278, val_acc: 0.5345
→ New best val_acc = 0.5345  (ckpt saved)
Epoch[1], val_loss: 3.5724, val_acc: 0.5228
Epoch[2], val_loss: 3.4807, val_acc: 0.5388
→ New best val_acc = 0.5388  (ckpt saved)
Epoch[3], val_loss: 3.1834, val_acc: 0.5322
Epoch[4], val_loss: 2.8663, val_acc: 0.4979
Epoch[5], val_loss: 2.7202, val_acc: 0.5042
Epoch[6], val_loss: 2.7075, val_acc: 0.4895
Epoch[7], val_loss: 2.6101, val_acc: 0.5229
Epoch[8], val_loss: 2.5294, val_acc: 0.5235
Epoch[9], val_loss: 2.5135, val_acc: 0.5053
Epoch[10], val_loss: 2.6025, val_acc: 0.5039
Epoch[11], val_loss: 2.5193, val_acc: 0.5089
Epoch[12], val_loss: 2.4263, val_acc: 0.4953
Epoch[13], val_loss: 2.4480, val_acc: 0.5023
Epoch[14], val_loss: 2.5064, val_acc: 0.5148
Epoch[15], val_loss: 2.4898, val_acc: 0.5025
Epoch[16], val_loss: 2.6109, val_acc: 0.5059
Epoch[17], val_loss: 2.5242, val_acc: 0.5275
Epoch[18], val_loss: 2.3363, val_acc: 0.5151
Epoch[19], val_loss: 2.6500, val_acc: 0.5128
Epoch[20], val_loss: 2.5429, val_acc: 0.5127
Epoch[21], val_loss: 2.4673, val_acc: 0.5084
Epoch[22], val_loss: 2.4870, val_acc: 0.5139
Early stopping at epoch 23 (no val_acc gain for 20 epochs).
```

In [ ]:
```
# ---- Curves ----------------------------------------------------------------
plot_accuracies(history_mobile)
```



In [ ]:
```
plot_losses(history_mobile)
```



In [ ]:
```python
from fvcore.nn import FlopCountAnalysis
# ———————— final test ————————————————————————————————
model_mobile.load_state_dict(torch.load('model_mobilenetv3_best.pth'))

# 6) Final evaluation on the test set
result_mobile_test = evaluate(model_mobile, test_dl)
print("MobileNetV3 Test result:", result_mobile_test)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops_mobile = FlopCountAnalysis(model_mobile.eval(), dummy_input)
gflops_mobile = flops_mobile.total() / 1e9
acc_mobile = result_mobile_test['val_acc']
efficiency_mobile = acc_mobile / gflops_mobile
print(f"MobileNetV3 GFLOPs: {gflops_mobile}")
print(f"MobileNetV3 Accuracy: {acc_mobile}")
print(f"MobileNetV3 Efficiency: {efficiency_mobile}")
```

```
MobileNetV3 Test result: {'val_loss': 3.3221442699432373, 'val_acc': 0.5354787707328796}
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::hardswish_ encountered 21 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::add_ encountered 10 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::hardsigmoid encountered 8 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::mul encountered 8 time(s)
WARNING:fvcore.nn.jit_analysis:Unsupported operator aten::dropout_ encountered 1 time(s)
MobileNetV3 GFLOPs: 0.014446376
MobileNetV3 Accuracy: 0.5354787707328796
MobileNetV3 Efficiency: 37.0666505380228
```

EfficientNet V2 S

In [ ]:
```python
from torchvision.models import efficientnet_v2_s, EfficientNet_V2_S_Weights
from torch import nn
import torch.nn.functional as F

class EfficientNetV2S(expression_model):
    def __init__(self, number_of_class: int, in_channel: int = 1):
        super().__init__()

        # ---- backbone (no weights) ----------------------------------------
        self.model = efficientnet_v2_s(weights=EfficientNet_V2_S_Weights)

        # ---- replace stem for 1-channel input -----------------------------
        # features[0] = ConvNormActivation(conv, bn, silu)
        old_conv = self.model.features[0][0]          # nn.Conv2d
        self.model.features[0][0] = nn.Conv2d(
            in_channels   = in_channel,
            out_channels  = old_conv.out_channels,    # 24
            kernel_size   = old_conv.kernel_size,
            stride        = old_conv.stride,
            padding       = old_conv.padding,
            bias=False
        )

        # ---- replace classifier head --------------------------------------
        in_feat = self.model.classifier[1].in_features  # 1280 for v2-s
        self.model.classifier[1] = nn.Linear(in_feat, number_of_class)
```

```
    # ------------------------------------------------------------------------
    def forward(self, x):
        return self.model(x)        # returns logits (batch, num_classes)

model_eff_2s = to_device(EfficientNetV2S(7,1),device)
print(count_params(model_eff_2s))
print(model_eff_2s)
```

```
20186023
EfficientNetV2S(
  (model): EfficientNet(
    (features): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(1, 24, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(24, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Sequential(
        (0): FusedMBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
              (1): BatchNorm2d(24, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.0, mode=row)
        )
        (1): FusedMBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
              (1): BatchNorm2d(24, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.005, mode=row)
        )
      )
      (2): Sequential(
        (0): FusedMBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(24, 96, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
              (1): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(96, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.01, mode=row)
        )
        (1): FusedMBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
              (1): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(192, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.015000000000000003, mode=row)
        )
        (2): FusedMBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
              (1): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(192, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.02, mode=row)
        )
        (3): FusedMBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
              (1): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(192, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
```

```
    (stochastic_depth): StochasticDepth(p=0.025, mode=row)
  )
)
(3): Sequential(
  (0): FusedMBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(48, 192, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.030000000000000006, mode=row)
  )
  (1): FusedMBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.035, mode=row)
  )
  (2): FusedMBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.04, mode=row)
  )
  (3): FusedMBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.045, mode=row)
  )
)
(4): Sequential(
  (0): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=256, bias=False)
        (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(256, 16, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(16, 256, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
```

```
  (stochastic_depth): StochasticDepth(p=0.05, mode=row)
)
(1): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=512, bias=False)
      (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.05500000000000001, mode=row)
)
(2): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=512, bias=False)
      (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.06000000000000001, mode=row)
)
(3): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=512, bias=False)
      (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.065, mode=row)
)
(4): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
```

```
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=512, bias=False)
        (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.07, mode=row)
  )
  (5): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=512, bias=False)
        (1): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.075, mode=row)
  )
)
(5): Sequential(
  (0): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(128, 768, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(768, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(768, 768, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=768, bias=False)
        (1): BatchNorm2d(768, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(768, 32, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(32, 768, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.08, mode=row)
  )
  (1): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=960, bias=False)
        (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
```

```
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.085, mode=row)
)
(2): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=960, bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.09, mode=row)
)
(3): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=960, bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.095, mode=row)
)
(4): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=960, bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
```

```
      (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.1, mode=row)
)
(5): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=960, bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.10500000000000001, mode=row)
)
(6): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=960, bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.11000000000000001, mode=row)
)
(7): MBConv(
  (block): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Conv2dNormActivation(
      (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=960, bias=False)
      (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (2): SqueezeExcitation(
      (avgpool): AdaptiveAvgPool2d(output_size=1)
      (fc1): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
      (fc2): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
      (activation): SiLU(inplace=True)
      (scale_activation): Sigmoid()
    )
    (3): Conv2dNormActivation(
      (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (stochastic_depth): StochasticDepth(p=0.11500000000000002, mode=row)
)
(8): MBConv(
  (block): Sequential(
```

```
          (0): Conv2dNormActivation(
            (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=960, bias=False)
            (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.12000000000000002, mode=row)
      )
    )
    (6): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=960, bias=False)
            (1): BatchNorm2d(960, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(960, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.125, mode=row)
      )
      (1): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.13, mode=row)
      )
      (2): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
```

```
            (1): Conv2dNormActivation(
              (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
              (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.135, mode=row)
        )
        (3): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
              (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.14, mode=row)
        )
        (4): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
              (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.14500000000000002, mode=row)
        )
        (5): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
              (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
```

```
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
          (activation): SiLU(inplace=True)
          (scale_activation): Sigmoid()
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (stochastic_depth): StochasticDepth(p=0.15, mode=row)
    )
    (6): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
          (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
          (activation): SiLU(inplace=True)
          (scale_activation): Sigmoid()
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (stochastic_depth): StochasticDepth(p=0.155, mode=row)
    )
    (7): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
          (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
          (activation): SiLU(inplace=True)
          (scale_activation): Sigmoid()
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (stochastic_depth): StochasticDepth(p=0.16, mode=row)
    )
    (8): MBConv(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
          (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          (2): SiLU(inplace=True)
        )
        (2): SqueezeExcitation(
          (avgpool): AdaptiveAvgPool2d(output_size=1)
          (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
          (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
```

```
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.165, mode=row)
      )
      (9): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.17, mode=row)
      )
      (10): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.175, mode=row)
      )
      (11): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
            (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.18, mode=row)
      )
      (12): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.185, mode=row)
      )
      (13): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.19, mode=row)
      )
      (14): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1536, bias=Fals
e)
            (1): BatchNorm2d(1536, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.195, mode=row)
      )
```

```
      )
      (7): Conv2dNormActivation(
        (0): Conv2d(256, 1280, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(1280, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=1)
    (classifier): Sequential(
      (0): Dropout(p=0.2, inplace=True)
      (1): Linear(in_features=1280, out_features=7, bias=True)
    )
  )
)
```

In [ ]:

```python
# ──────────────────────────────────────────────────────────────
#  EfficientNetV2-S  →  optimised training loop (FER-2013)
# ──────────────────────────────────────────────────────────────
import math, torch

# ── Balanced Hyper-parameters ──────────────────────────────────
epochs_eff_2s        = 100        # Fewer epochs since using ImageNet weights
batch_size_eff_2s    = 64         # Larger batch size for smaller model
base_lr_eff_2s       = 1e-4       # Lower LR since using pretrained weights
max_lr_eff_2s        = 1e-3       # Lower max LR for fine-tuning
weight_decay_eff_2s  = 0.05       # Reduced weight decay for pretrained model
label_smooth_eff_2s  = 0.1
grad_clip_norm       = 1.0        # Lower grad clip for stable fine-tuning
patience_eff_2s      = 10         # Shorter patience since using pretrained

# ❷  Optimiser – parameter-wise weight-decay split with layer-wise params
decay, no_decay = [], []
for n, p in model_eff_2s.named_parameters():
    if not p.requires_grad: continue
    # Skip weight decay for biases, LayerNorm and position embeddings
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_eff_2s = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_eff_2s},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_eff_2s, betas=(0.9, 0.95), eps=1e-8  # Lower beta2 for fine-tuning
)

# ❸  Cosine scheduler with shorter warmup for pretrained model
sched_eff_2s = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_eff_2s,
    max_lr          = max_lr_eff_2s,
    epochs          = epochs_eff_2s,
    steps_per_epoch = len(train_dl),
    pct_start       = 0.05,       # 5% warmup period for pretrained
    div_factor      = 10,         # Reduced LR range for fine-tuning
    final_div_factor= 1000
)

# ❻  Training loop with early-stop and checkpoint
best_val_acc_eff_2s   = 0.0
epochs_no_imp_eff_2s  = 0
history_eff_2s        = []

for epoch in range(epochs_eff_2s):
    # ---- TRAIN ------------------------------------------------------
    model_eff_2s.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_2s.training_step(batch)       # CE + smooth inside
        train_losses.append(loss.detach())
        loss.backward()

        # clip gradients *before* stepping
        torch.nn.utils.clip_grad_norm_(model_eff_2s.parameters(), grad_clip_norm)

        optimizer_eff_2s.step()
        optimizer_eff_2s.zero_grad(set_to_none=True)  # more efficient zero_grad
        sched_eff_2s.step()

    # ---- VALIDATE ---------------------------------------------------
    result = evaluate(model_eff_2s, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_2s.epoch_end(epoch, result)                  # prints nicely
    history_eff_2s.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------
    val_acc = result['val_acc']
```

```
    if val_acc > best_val_acc_eff_2s:
        best_val_acc_eff_2s, epochs_no_imp_eff_2s = val_acc, 0
        torch.save(model_eff_2s.state_dict(), 'model_efficientnetv2s_best.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_eff_2s += 1
        if epochs_no_imp_eff_2s >= patience_eff_2s:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_eff_2s} epochs).")
            break
```

```
Epoch[0], val_loss: 1.8087, val_acc: 0.2583
→ New best val_acc = 0.2583  (ckpt saved)
Epoch[1], val_loss: 1.5362, val_acc: 0.4176
→ New best val_acc = 0.4176  (ckpt saved)
Epoch[2], val_loss: 1.3866, val_acc: 0.5034
→ New best val_acc = 0.5034  (ckpt saved)
Epoch[3], val_loss: 1.2625, val_acc: 0.5426
→ New best val_acc = 0.5426  (ckpt saved)
Epoch[4], val_loss: 1.1977, val_acc: 0.5594
→ New best val_acc = 0.5594  (ckpt saved)
Epoch[5], val_loss: 1.2112, val_acc: 0.5849
→ New best val_acc = 0.5849  (ckpt saved)
Epoch[6], val_loss: 1.3941, val_acc: 0.5819
Epoch[7], val_loss: 1.5113, val_acc: 0.5745
Epoch[8], val_loss: 1.5765, val_acc: 0.5881
→ New best val_acc = 0.5881  (ckpt saved)
Epoch[9], val_loss: 1.7666, val_acc: 0.5775
Epoch[10], val_loss: 1.7250, val_acc: 0.5667
Epoch[11], val_loss: 1.9160, val_acc: 0.6024
→ New best val_acc = 0.6024  (ckpt saved)
Epoch[12], val_loss: 2.1848, val_acc: 0.5880
Epoch[13], val_loss: 2.0978, val_acc: 0.5821
Epoch[14], val_loss: 1.8694, val_acc: 0.5840
Epoch[15], val_loss: 2.0514, val_acc: 0.5937
Epoch[16], val_loss: 2.1013, val_acc: 0.5987
Epoch[17], val_loss: 2.1437, val_acc: 0.6039
→ New best val_acc = 0.6039  (ckpt saved)
Epoch[18], val_loss: 2.2775, val_acc: 0.5893
Epoch[19], val_loss: 2.2063, val_acc: 0.5814
Epoch[20], val_loss: 2.2690, val_acc: 0.6007
Epoch[21], val_loss: 2.4786, val_acc: 0.5784
Epoch[22], val_loss: 2.2103, val_acc: 0.5807
Epoch[23], val_loss: 2.3709, val_acc: 0.5898
Epoch[24], val_loss: 2.2549, val_acc: 0.6023
Epoch[25], val_loss: 2.2919, val_acc: 0.5856
Epoch[26], val_loss: 2.2171, val_acc: 0.6045
→ New best val_acc = 0.6045  (ckpt saved)
Epoch[27], val_loss: 2.4640, val_acc: 0.5957
Epoch[28], val_loss: 2.5395, val_acc: 0.5892
Epoch[29], val_loss: 2.5321, val_acc: 0.5892
Epoch[30], val_loss: 2.6085, val_acc: 0.5850
Epoch[31], val_loss: 2.4982, val_acc: 0.5985
Epoch[32], val_loss: 2.6596, val_acc: 0.5951
Epoch[33], val_loss: 2.4556, val_acc: 0.5906
Epoch[34], val_loss: 2.7008, val_acc: 0.5827
Epoch[35], val_loss: 2.6227, val_acc: 0.5957
Epoch[36], val_loss: 2.5774, val_acc: 0.5913
Early stopping at epoch 37 (no val_acc gain for 10 epochs).
```

In [ ]:

```
# Further fine tuning with reduced learning rates and epochs
model_eff_2s.load_state_dict(torch.load('model_efficientnetv2s_best.pth'))

# ── Hyper-parameters for further fine-tuning ──────────────────────────
epochs_eff_2s        = 50         # Reduced epochs for final fine-tuning
batch_size_eff_2s    = 32         # Smaller batch size for better generalization
base_lr_eff_2s       = 5e-5       # Much lower LR for final fine-tuning
max_lr_eff_2s        = 5e-4       # Much lower max LR for final fine-tuning
weight_decay_eff_2s  = 0.02       # Further reduced weight decay
label_smooth_eff_2s  = 0.05       # Reduced label smoothing
grad_clip_norm       = 0.5        # Tighter gradient clipping
patience_eff_2s      = 15         # Increased patience for finding subtle improvements

# 🔹 Optimiser with reduced learning rates
decay, no_decay = [], []
for n, p in model_eff_2s.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_eff_2s = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_eff_2s},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_eff_2s, betas=(0.9, 0.99), eps=1e-8  # Higher beta2 for stability
```

```
)

# 6️⃣  Gentler learning rate schedule
sched_eff_2s = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_eff_2s,
    max_lr          = max_lr_eff_2s,
    epochs          = epochs_eff_2s,
    steps_per_epoch = len(train_dl),
    pct_start       = 0.1,        # Longer warmup
    div_factor      = 5,          # Narrower LR range
    final_div_factor= 100         # Less aggressive final LR reduction
)

# 7️⃣  Training loop with early-stop and checkpoint
best_val_acc_eff_2s   = 0.0
epochs_no_imp_eff_2s  = 0
history_eff_2s        = []

for epoch in range(epochs_eff_2s):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_2s.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_2s.training_step(batch)
        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_2s.parameters(), grad_clip_norm)

        optimizer_eff_2s.step()
        optimizer_eff_2s.zero_grad(set_to_none=True)
        sched_eff_2s.step()

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_2s, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_2s.epoch_end(epoch, result)
    history_eff_2s.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_eff_2s:
        best_val_acc_eff_2s, epochs_no_imp_eff_2s = val_acc, 0
        torch.save(model_eff_2s.state_dict(), 'model_efficientnetv2s_final.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (final ckpt saved)")
    else:
        epochs_no_imp_eff_2s += 1
        if epochs_no_imp_eff_2s >= patience_eff_2s:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_eff_2s} epochs).")
            break
```

```
Epoch[0], val_loss: 2.3609, val_acc: 0.6098
→ New best val_acc = 0.6098  (final ckpt saved)
Epoch[1], val_loss: 2.6805, val_acc: 0.6077
Epoch[2], val_loss: 2.9783, val_acc: 0.6033
Epoch[3], val_loss: 2.9630, val_acc: 0.6063
Epoch[4], val_loss: 2.8596, val_acc: 0.6048
Epoch[5], val_loss: 2.6409, val_acc: 0.5949
Epoch[6], val_loss: 2.8097, val_acc: 0.6051
Epoch[7], val_loss: 2.7874, val_acc: 0.6099
→ New best val_acc = 0.6099  (final ckpt saved)
Epoch[8], val_loss: 2.7873, val_acc: 0.6017
Epoch[9], val_loss: 2.9972, val_acc: 0.5975
Epoch[10], val_loss: 2.8852, val_acc: 0.6017
Epoch[11], val_loss: 3.0472, val_acc: 0.6003
Epoch[12], val_loss: 3.0036, val_acc: 0.6047
Epoch[13], val_loss: 3.0520, val_acc: 0.6030
Epoch[14], val_loss: 3.1021, val_acc: 0.5959
Epoch[15], val_loss: 3.0601, val_acc: 0.5977
Epoch[16], val_loss: 3.2070, val_acc: 0.6004
Epoch[17], val_loss: 3.0130, val_acc: 0.5997
Epoch[18], val_loss: 3.1396, val_acc: 0.5997
Epoch[19], val_loss: 3.1742, val_acc: 0.6000
Epoch[20], val_loss: 3.2227, val_acc: 0.6047
Epoch[21], val_loss: 3.0973, val_acc: 0.6020
Epoch[22], val_loss: 3.1488, val_acc: 0.6003
Early stopping at epoch 23 (no val_acc gain for 15 epochs).
```

In [ ]:
```
# Load best model checkpoint and add regularization for fine-tuning
model_eff_2s.load_state_dict(torch.load('model_efficientnetv2s_final.pth'))

# —— Hyper-parameters for regularized fine-tuning ————————————————
epochs_eff_2s        = 80         # Reduced epochs for final fine-tuning
batch_size_eff_2s    = 16         # Even smaller batch size for better generalization
base_lr_eff_2s       = 1e-5       # Very low LR for careful fine-tuning
```

```
max_lr_eff_2s        = 1e-4       # Very low max LR for careful fine-tuning
weight_decay_eff_2s  = 0.05       # Increased weight decay for regularization
label_smooth_eff_2s  = 0.1        # Increased label smoothing
grad_clip_norm       = 0.3        # Stricter gradient clipping
patience_eff_2s      = 20         # More patience for finding subtle improvements
dropout_rate         = 0.3        # Add dropout for regularization

# 2️⃣ Optimiser with reduced learning rates
decay, no_decay = [], []
for n, p in model_eff_2s.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_eff_2s = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_eff_2s},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_eff_2s, betas=(0.9, 0.999), eps=1e-8
)

# 3️⃣ Very conservative learning rate schedule
sched_eff_2s = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_eff_2s,
    max_lr          = max_lr_eff_2s,
    epochs          = epochs_eff_2s,
    steps_per_epoch = len(train_dl),
    pct_start       = 0.2,        # Even longer warmup
    div_factor      = 10,         # Wider LR range for careful exploration
    final_div_factor= 1000        # More aggressive final LR reduction
)

# 5️⃣ Training loop with early-stop, checkpoint and mixup augmentation
best_val_acc_eff_2s   = 0.0
epochs_no_imp_eff_2s  = 0
history_eff_2s        = []

for epoch in range(epochs_eff_2s):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_2s.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_2s.training_step(batch)
        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_2s.parameters(), grad_clip_norm)

        optimizer_eff_2s.step()
        optimizer_eff_2s.zero_grad(set_to_none=True)
        sched_eff_2s.step()

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_2s, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_2s.epoch_end(epoch, result)
    history_eff_2s.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_eff_2s:
        best_val_acc_eff_2s, epochs_no_imp_eff_2s = val_acc, 0
        torch.save(model_eff_2s.state_dict(), 'model_efficientnetv2s_final_tuned.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (final ckpt saved)")
    else:
        epochs_no_imp_eff_2s += 1
        if epochs_no_imp_eff_2s >= patience_eff_2s:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_eff_2s} epochs).")
            break
```

```
Epoch[0], val_loss: 2.8131, val_acc: 0.6045
→ New best val_acc = 0.6045  (final ckpt saved)
Epoch[1], val_loss: 2.8050, val_acc: 0.6109
→ New best val_acc = 0.6109  (final ckpt saved)
Epoch[2], val_loss: 2.7945, val_acc: 0.6126
→ New best val_acc = 0.6126  (final ckpt saved)
Epoch[3], val_loss: 2.8216, val_acc: 0.6106
Epoch[4], val_loss: 2.8579, val_acc: 0.6090
Epoch[5], val_loss: 2.8610, val_acc: 0.6098
Epoch[6], val_loss: 2.9319, val_acc: 0.6081
Epoch[7], val_loss: 2.9410, val_acc: 0.6068
Epoch[8], val_loss: 2.9832, val_acc: 0.6098
Epoch[9], val_loss: 3.0507, val_acc: 0.6121
Epoch[10], val_loss: 3.1054, val_acc: 0.6101
Epoch[11], val_loss: 3.1561, val_acc: 0.6111
Epoch[12], val_loss: 3.1960, val_acc: 0.6146
→ New best val_acc = 0.6146  (final ckpt saved)
Epoch[13], val_loss: 3.2048, val_acc: 0.6144
Epoch[14], val_loss: 3.2162, val_acc: 0.6141
Epoch[15], val_loss: 3.2248, val_acc: 0.6204
→ New best val_acc = 0.6204  (final ckpt saved)
Epoch[16], val_loss: 3.2831, val_acc: 0.6120
Epoch[17], val_loss: 3.2900, val_acc: 0.6119
Epoch[18], val_loss: 3.4008, val_acc: 0.6149
Epoch[19], val_loss: 3.3614, val_acc: 0.6160
Epoch[20], val_loss: 3.3239, val_acc: 0.6093
Epoch[21], val_loss: 3.3722, val_acc: 0.6159
Epoch[22], val_loss: 3.4538, val_acc: 0.6184
Epoch[23], val_loss: 3.5112, val_acc: 0.6212
→ New best val_acc = 0.6212  (final ckpt saved)
Epoch[24], val_loss: 3.4568, val_acc: 0.6136
Epoch[25], val_loss: 3.5414, val_acc: 0.6157
Epoch[26], val_loss: 3.5330, val_acc: 0.6162
Epoch[27], val_loss: 3.5481, val_acc: 0.6184
Epoch[28], val_loss: 3.5535, val_acc: 0.6093
Epoch[29], val_loss: 3.5103, val_acc: 0.6085
Epoch[30], val_loss: 3.5507, val_acc: 0.6186
Epoch[31], val_loss: 3.5473, val_acc: 0.6148
Epoch[32], val_loss: 3.5389, val_acc: 0.6183
Epoch[33], val_loss: 3.5753, val_acc: 0.6200
Epoch[34], val_loss: 3.5782, val_acc: 0.6231
→ New best val_acc = 0.6231  (final ckpt saved)
Epoch[35], val_loss: 3.5749, val_acc: 0.6156
Epoch[36], val_loss: 3.5243, val_acc: 0.6191
Epoch[37], val_loss: 3.5913, val_acc: 0.6196
Epoch[38], val_loss: 3.5735, val_acc: 0.6171
Epoch[39], val_loss: 3.5578, val_acc: 0.6179
Epoch[40], val_loss: 3.5462, val_acc: 0.6201
Epoch[41], val_loss: 3.5613, val_acc: 0.6246
→ New best val_acc = 0.6246  (final ckpt saved)
Epoch[42], val_loss: 3.5493, val_acc: 0.6189
Epoch[43], val_loss: 3.5873, val_acc: 0.6169
Epoch[44], val_loss: 3.5935, val_acc: 0.6189
Epoch[45], val_loss: 3.5939, val_acc: 0.6229
Epoch[46], val_loss: 3.6065, val_acc: 0.6264
→ New best val_acc = 0.6264  (final ckpt saved)
Epoch[47], val_loss: 3.6128, val_acc: 0.6236
Epoch[48], val_loss: 3.5960, val_acc: 0.6239
Epoch[49], val_loss: 3.6178, val_acc: 0.6231
Epoch[50], val_loss: 3.6303, val_acc: 0.6221
Epoch[51], val_loss: 3.6547, val_acc: 0.6231
Epoch[52], val_loss: 3.6722, val_acc: 0.6244
Epoch[53], val_loss: 3.6699, val_acc: 0.6247
Epoch[54], val_loss: 3.6876, val_acc: 0.6254
Epoch[55], val_loss: 3.6819, val_acc: 0.6249
Epoch[56], val_loss: 3.6978, val_acc: 0.6282
→ New best val_acc = 0.6282  (final ckpt saved)
Epoch[57], val_loss: 3.7049, val_acc: 0.6294
→ New best val_acc = 0.6294  (final ckpt saved)
Epoch[58], val_loss: 3.6859, val_acc: 0.6302
→ New best val_acc = 0.6302  (final ckpt saved)
Epoch[59], val_loss: 3.6678, val_acc: 0.6271
Epoch[60], val_loss: 3.6742, val_acc: 0.6282
Epoch[61], val_loss: 3.6877, val_acc: 0.6274
Epoch[62], val_loss: 3.6859, val_acc: 0.6279
Epoch[63], val_loss: 3.6940, val_acc: 0.6294
Epoch[64], val_loss: 3.6827, val_acc: 0.6282
Epoch[65], val_loss: 3.6746, val_acc: 0.6292
Epoch[66], val_loss: 3.6818, val_acc: 0.6295
Epoch[67], val_loss: 3.6841, val_acc: 0.6279
Epoch[68], val_loss: 3.6796, val_acc: 0.6287
Epoch[69], val_loss: 3.6787, val_acc: 0.6297
Epoch[70], val_loss: 3.6883, val_acc: 0.6282
Epoch[71], val_loss: 3.6824, val_acc: 0.6307
→ New best val_acc = 0.6307  (final ckpt saved)
Epoch[72], val_loss: 3.6981, val_acc: 0.6287
```

```
Epoch[73], val_loss: 3.6824, val_acc: 0.6299
Epoch[74], val_loss: 3.6765, val_acc: 0.6284
Epoch[75], val_loss: 3.6967, val_acc: 0.6289
Epoch[76], val_loss: 3.6848, val_acc: 0.6277
Epoch[77], val_loss: 3.6959, val_acc: 0.6284
Epoch[78], val_loss: 3.6817, val_acc: 0.6289
Epoch[79], val_loss: 3.6874, val_acc: 0.6297
```

In [ ]:
```python
from fvcore.nn import FlopCountAnalysis
# ———————— final test ————————————————————————————————
# model_eff_2s.load_state_dict(torch.load('model_efficientnetv2s_best.pth'))
# model_eff_2s.load_state_dict(torch.load('model_efficientnetv2s_final.pth'))
model_eff_2s.load_state_dict(torch.load('model_efficientnetv2s_final_tuned.pth'))


# 6) Final evaluation on the test set
result_eff_test = evaluate(model_eff_2s, test_dl)
print("EfficientNetV2-S Test result:", result_eff_test)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops_eff = FlopCountAnalysis(model_eff_2s.eval(), dummy_input)
gflops_eff = flops_eff.total() / 1e9
acc_eff = result_eff_test['val_acc']
efficiency_eff = acc_eff / gflops_eff
print(f"EfficientNetV2-S GFLOPs: {gflops_eff}")
print(f"EfficientNetV2-S Accuracy: {acc_eff}")
print(f"EfficientNetV2-S Efficiency: {efficiency_eff}")
```

```
EfficientNetV2-S Test result: {'val_loss': 3.504228115081787, 'val_acc': 0.6292942762374878}
Unsupported operator aten::silu_ encountered 102 time(s)
Unsupported operator aten::add_ encountered 35 time(s)
Unsupported operator aten::sigmoid encountered 30 time(s)
Unsupported operator aten::mul encountered 30 time(s)
Unsupported operator aten::dropout_ encountered 1 time(s)
The following submodules of the model were never called during the trace of the graph. They may be unused, or th
ey were accessed by direct calls to .forward() or via other python methods. In the latter case they will have ze
ros for statistics, though their statistics will still contribute to their parent calling module.
model.features.1.0.stochastic_depth, model.features.1.1.stochastic_depth, model.features.2.0.stochastic_depth, m
odel.features.2.1.stochastic_depth, model.features.2.2.stochastic_depth, model.features.2.3.stochastic_depth, mo
del.features.3.0.stochastic_depth, model.features.3.1.stochastic_depth, model.features.3.2.stochastic_depth, mod
el.features.3.3.stochastic_depth, model.features.4.0.stochastic_depth, model.features.4.1.stochastic_depth, mode
l.features.4.2.stochastic_depth, model.features.4.3.stochastic_depth, model.features.4.4.stochastic_depth, mode
l.features.4.5.stochastic_depth, model.features.5.0.stochastic_depth, model.features.5.1.stochastic_depth, mode
l.features.5.2.stochastic_depth, model.features.5.3.stochastic_depth, model.features.5.4.stochastic_depth, mode
l.features.5.5.stochastic_depth, model.features.5.6.stochastic_depth, model.features.5.7.stochastic_depth, mode
l.features.5.8.stochastic_depth, model.features.6.0.stochastic_depth, model.features.6.1.stochastic_depth, mode
l.features.6.10.stochastic_depth, model.features.6.11.stochastic_depth, model.features.6.12.stochastic_depth, mo
del.features.6.13.stochastic_depth, model.features.6.14.stochastic_depth, model.features.6.2.stochastic_depth, m
odel.features.6.3.stochastic_depth, model.features.6.4.stochastic_depth, model.features.6.5.stochastic_depth, mo
del.features.6.6.stochastic_depth, model.features.6.7.stochastic_depth, model.features.6.8.stochastic_depth, mod
el.features.6.9.stochastic_depth
EfficientNetV2-S GFLOPs: 0.156170176
EfficientNetV2-S Accuracy: 0.6292942762374878
EfficientNetV2-S Efficiency: 4.029541954524581
```

EfficientNet b0 on standard dataset

In [ ]:
```python
from torchvision.models import efficientnet_b0, EfficientNet_B0_Weights
from torch import nn
import torch.nn.functional as F

class EfficientNetB0(expression_model):
    def __init__(self, number_of_class: int, in_channel: int = 1):
        super().__init__()

        # ---- backbone (no weights) ----------------------------------------
        self.model = efficientnet_b0(weights=EfficientNet_B0_Weights.DEFAULT)

        # ---- replace stem for 1-channel input -----------------------------
        # features[0] = ConvNormActivation(conv, bn, silu)
        old_conv = self.model.features[0][0]          # nn.Conv2d
        self.model.features[0][0] = nn.Conv2d(
            in_channels   = in_channel,
            out_channels  = old_conv.out_channels,    # 32
            kernel_size   = old_conv.kernel_size,
            stride        = old_conv.stride,
            padding       = old_conv.padding,
            bias=False
        )

        # ---- replace classifier head --------------------------------------
        in_feat = self.model.classifier[1].in_features  # 1280 for b0
        self.model.classifier[1] = nn.Linear(in_feat, number_of_class)
```

```
    # ------------------------------------------------------------------------
    def forward(self, x):
        return self.model(x)        # returns logits (batch, num_classes)

model_eff_b0 = to_device(EfficientNetB0(7,1),device)
print(count_params(model_eff_b0))
print(model_eff_b0)
```

```
4015939
EfficientNetB0(
  (model): EfficientNet(
    (features): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
              (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(32, 8, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(8, 32, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (2): Conv2dNormActivation(
              (0): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.0, mode=row)
        )
      )
      (2): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(16, 96, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=96, bias=False)
              (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(96, 4, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(4, 96, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(96, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.0125, mode=row)
        )
        (1): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(144, 144, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=144, bias=False)
              (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(144, 6, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(6, 144, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(144, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.025, mode=row)
        )
      )
```

```
(3): Sequential(
  (0): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(144, 144, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), groups=144, bias=False)
        (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(144, 6, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(6, 144, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(144, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.037500000000000006, mode=row)
  )
  (1): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(240, 240, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=240, bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(240, 10, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(10, 240, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(240, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.05, mode=row)
  )
)
(4): Sequential(
  (0): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(240, 240, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=240, bias=False)
        (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(240, 10, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(10, 240, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(240, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.0625, mode=row)
  )
  (1): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=480, bias=False)
        (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(480, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.07500000000000001, mode=row)
  )
  (2): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=480, bias=False)
        (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(480, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.08750000000000001, mode=row)
  )
)
(5): Sequential(
  (0): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(480, 480, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=480, bias=False)
        (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (2): SqueezeExcitation(
        (avgpool): AdaptiveAvgPool2d(output_size=1)
        (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
        (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU(inplace=True)
        (scale_activation): Sigmoid()
      )
      (3): Conv2dNormActivation(
        (0): Conv2d(480, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (stochastic_depth): StochasticDepth(p=0.1, mode=row)
  )
  (1): MBConv(
    (block): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=672, bias=False)
        (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
```

```
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.1125, mode=row)
        )
        (2): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=672, bias=False)
              (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.125, mode=row)
        )
      )
      (6): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), groups=672, bias=False)
              (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(672, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.1375, mode=row)
        )
        (1): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=1152, bias=Fals
e)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
```

```
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.15000000000000002, mode=row)
        )
        (2): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=1152, bias=Fals
e)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.1625, mode=row)
        )
        (3): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=1152, bias=Fals
e)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.17500000000000002, mode=row)
        )
      )
      (7): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1152, 1152, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=1152, bias=Fals
e)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
```

```
            (3): Conv2dNormActivation(
              (0): Conv2d(1152, 320, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.1875, mode=row)
        )
      )
      (8): Conv2dNormActivation(
        (0): Conv2d(320, 1280, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(1280, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): SiLU(inplace=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=1)
    (classifier): Sequential(
      (0): Dropout(p=0.2, inplace=True)
      (1): Linear(in_features=1280, out_features=7, bias=True)
    )
  )
)
```

In [ ]:

```python
# ──────────────────────────────────────────────────────────────
#  EfficientNet-B0  →  optimised training loop (FER-2013)
# ──────────────────────────────────────────────────────────────
import math, torch

# ── Balanced Hyper-parameters ──────────────────────────────────
epochs_eff_b0        = 300        # More epochs for training from scratch
batch_size_eff_b0    = 64         # Larger batch size for smaller model
base_lr_eff_b0       = 1e-3       # Higher LR since training from scratch
max_lr_eff_b0        = 1e-2       # Higher max LR for full training
weight_decay_eff_b0  = 0.1        # Standard weight decay
label_smooth_eff_b0  = 0.1
grad_clip_norm       = 2.0        # Higher grad clip for from-scratch training
patience_eff_b0      = 20         # Longer patience for full training

# ❷  Optimiser – parameter-wise weight-decay split with layer-wise params
decay, no_decay = [], []
for n, p in model_eff_b0.named_parameters():
    if not p.requires_grad: continue
    # Skip weight decay for biases, LayerNorm and position embeddings
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_eff_b0 = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_eff_b0},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_eff_b0, betas=(0.9, 0.999), eps=1e-8  # Standard betas
)

# ❸  Cosine scheduler with longer warmup for from-scratch training
sched_eff_b0 = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_eff_b0,
    max_lr          = max_lr_eff_b0,
    epochs          = epochs_eff_b0,
    steps_per_epoch = len(train_dl),
    pct_start       = 0.1,        # 10% warmup period for from-scratch
    div_factor      = 10,         # Standard LR range
    final_div_factor= 1000
)

# ❺  Training loop with early-stop and checkpoint
best_val_acc_eff_b0   = 0.0
epochs_no_imp_eff_b0  = 0
history_eff_b0        = []

for epoch in range(epochs_eff_b0):
    # ---- TRAIN ------------------------------------------------------
    model_eff_b0.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_b0.training_step(batch)       # CE + smooth inside
        train_losses.append(loss.detach())
        loss.backward()

        # clip gradients *before* stepping
        torch.nn.utils.clip_grad_norm_(model_eff_b0.parameters(), grad_clip_norm)

        optimizer_eff_b0.step()
        optimizer_eff_b0.zero_grad(set_to_none=True)  # more efficient zero_grad
        sched_eff_b0.step()

    # ---- VALIDATE ---------------------------------------------------
```

```
    result = evaluate(model_eff_b0, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0.epoch_end(epoch, result)                     # prints nicely
    history_eff_b0.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_eff_b0:
        best_val_acc_eff_b0, epochs_no_imp_eff_b0 = val_acc, 0
        torch.save(model_eff_b0.state_dict(), 'model_efficientnetb0.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_eff_b0 += 1
        if epochs_no_imp_eff_b0 >= patience_eff_b0:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_eff_b0} epochs).")
            break
```

```
Epoch[0], val_loss: 1.6238, val_acc: 0.3914
→ New best val_acc = 0.3914  (ckpt saved)
Epoch[1], val_loss: 1.3487, val_acc: 0.4793
→ New best val_acc = 0.4793  (ckpt saved)
Epoch[2], val_loss: 1.3287, val_acc: 0.5084
→ New best val_acc = 0.5084  (ckpt saved)
Epoch[3], val_loss: 1.3026, val_acc: 0.5352
→ New best val_acc = 0.5352  (ckpt saved)
Epoch[4], val_loss: 1.3096, val_acc: 0.5317
Epoch[5], val_loss: 1.3652, val_acc: 0.5321
Epoch[6], val_loss: 1.3718, val_acc: 0.5454
→ New best val_acc = 0.5454  (ckpt saved)
Epoch[7], val_loss: 1.4133, val_acc: 0.5273
Epoch[8], val_loss: 1.3930, val_acc: 0.5367
Epoch[9], val_loss: 1.5022, val_acc: 0.5362
Epoch[10], val_loss: 1.4168, val_acc: 0.5569
→ New best val_acc = 0.5569  (ckpt saved)
Epoch[11], val_loss: 1.4842, val_acc: 0.5379
Epoch[12], val_loss: 1.4634, val_acc: 0.5493
Epoch[13], val_loss: 1.4187, val_acc: 0.5410
Epoch[14], val_loss: 1.4256, val_acc: 0.5529
Epoch[15], val_loss: 1.4060, val_acc: 0.5528
Epoch[16], val_loss: 1.3823, val_acc: 0.5542
Epoch[17], val_loss: 1.4725, val_acc: 0.5516
Epoch[18], val_loss: 1.3879, val_acc: 0.5623
→ New best val_acc = 0.5623  (ckpt saved)
Epoch[19], val_loss: 1.3397, val_acc: 0.5653
→ New best val_acc = 0.5653  (ckpt saved)
Epoch[20], val_loss: 1.3700, val_acc: 0.5393
Epoch[21], val_loss: 1.3514, val_acc: 0.5407
Epoch[22], val_loss: 1.3997, val_acc: 0.5418
Epoch[23], val_loss: 1.3717, val_acc: 0.5465
Epoch[24], val_loss: 1.3827, val_acc: 0.5466
Epoch[25], val_loss: 1.3300, val_acc: 0.5588
Epoch[26], val_loss: 1.4568, val_acc: 0.5184
Epoch[27], val_loss: 1.3741, val_acc: 0.5462
Epoch[28], val_loss: 1.3510, val_acc: 0.5621
Epoch[29], val_loss: 1.3714, val_acc: 0.5539
Epoch[30], val_loss: 1.3632, val_acc: 0.5542
Epoch[31], val_loss: 1.3798, val_acc: 0.5521
Epoch[32], val_loss: 1.3669, val_acc: 0.5724
→ New best val_acc = 0.5724  (ckpt saved)
Epoch[33], val_loss: 1.4021, val_acc: 0.5488
Epoch[34], val_loss: 1.4302, val_acc: 0.5511
Epoch[35], val_loss: 1.3364, val_acc: 0.5769
→ New best val_acc = 0.5769  (ckpt saved)
Epoch[36], val_loss: 1.4668, val_acc: 0.5578
Epoch[37], val_loss: 1.4494, val_acc: 0.5530
Epoch[38], val_loss: 1.5101, val_acc: 0.5515
Epoch[39], val_loss: 1.5253, val_acc: 0.5491
Epoch[40], val_loss: 1.4154, val_acc: 0.5606
Epoch[41], val_loss: 1.6389, val_acc: 0.5258
Epoch[42], val_loss: 1.4502, val_acc: 0.5710
Epoch[43], val_loss: 1.4905, val_acc: 0.5675
Epoch[44], val_loss: 1.5542, val_acc: 0.5402
Epoch[45], val_loss: 1.6242, val_acc: 0.5492
Epoch[46], val_loss: 1.6525, val_acc: 0.5069
Epoch[47], val_loss: 1.5844, val_acc: 0.5574
Epoch[48], val_loss: 1.5685, val_acc: 0.5419
Epoch[49], val_loss: 1.4697, val_acc: 0.5459
Epoch[50], val_loss: 1.5904, val_acc: 0.5800
→ New best val_acc = 0.5800  (ckpt saved)
Epoch[51], val_loss: 1.5924, val_acc: 0.5550
Epoch[52], val_loss: 1.6098, val_acc: 0.5621
Epoch[53], val_loss: 1.5008, val_acc: 0.5719
Epoch[54], val_loss: 1.5673, val_acc: 0.5539
Epoch[55], val_loss: 1.5207, val_acc: 0.5514
Epoch[56], val_loss: 1.6318, val_acc: 0.5628
Epoch[57], val_loss: 1.5563, val_acc: 0.5534
Epoch[58], val_loss: 1.7341, val_acc: 0.5388
Epoch[59], val_loss: 1.6056, val_acc: 0.5734
Epoch[60], val_loss: 1.7056, val_acc: 0.5460
Epoch[61], val_loss: 1.5925, val_acc: 0.5525
Epoch[62], val_loss: 1.5633, val_acc: 0.5624
Epoch[63], val_loss: 1.5314, val_acc: 0.5728
Epoch[64], val_loss: 1.5241, val_acc: 0.5605
Epoch[65], val_loss: 1.6367, val_acc: 0.5422
Epoch[66], val_loss: 1.4702, val_acc: 0.5696
Epoch[67], val_loss: 1.5497, val_acc: 0.5737
Epoch[68], val_loss: 1.6085, val_acc: 0.5540
Epoch[69], val_loss: 1.6647, val_acc: 0.5445
Epoch[70], val_loss: 1.6979, val_acc: 0.5472
Early stopping at epoch 71 (no val_acc gain for 20 epochs).
```

Tuned 1

In [ ]:

```python
# Further fine tuning with reduced learning rates and epochs
model_eff_b0.load_state_dict(torch.load('model_efficientnetb0.pth'))

# —— Hyper-parameters for further fine-tuning ————————————————————
epochs_ft        = 150         # One-third of initial epochs for fine-tuning
batch_size_ft    = 32         # Half the initial batch size for better generalization
base_lr_ft       = 1e-4       # One-tenth of initial base LR for fine-tuning
max_lr_ft        = 1e-3       # One-tenth of initial max LR for fine-tuning
weight_decay_ft  = 0.05       # Half the initial weight decay
label_smooth_ft  = 0.05       # Half the initial label smoothing
grad_clip_norm   = 1.0        # Half the initial gradient clipping
patience_ft      = 20         # Slightly reduced patience for fine-tuning

# ❷  Optimiser with reduced learning rates
decay, no_decay = [], []
for n, p in model_eff_b0.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_ft = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_ft},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_ft, betas=(0.9, 0.99), eps=1e-8  # Higher beta2 for stability
)

# ❸  Learning rate schedule using RLROP
sched_ft = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_ft,
    mode='max',           # Monitor validation accuracy
    factor=0.5,          # Reduce LR by half when plateauing
    patience=5,          # Wait 5 epochs before reducing LR
    verbose=True,        # Print message when LR changes
    min_lr=1e-7         # Don't reduce LR below this
)

# ❻  Training loop with early-stop and checkpoint
best_val_acc_ft   = 0.0
epochs_no_imp_ft  = 0
history_ft        = []

for epoch in range(epochs_ft):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_b0.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_b0.training_step(batch)
        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0.parameters(), grad_clip_norm)

        optimizer_ft.step()
        optimizer_ft.zero_grad(set_to_none=True)

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_b0, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0.epoch_end(epoch, result)
    history_ft.append(result)

    # Update learning rate based on validation accuracy
    sched_ft.step(result['val_acc'])

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_ft:
        best_val_acc_ft, epochs_no_imp_ft = val_acc, 0
        torch.save(model_eff_b0.state_dict(), 'model_efficientnetb0_tuned.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_ft += 1
        if epochs_no_imp_ft >= patience_ft:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_ft} epochs).")
            break
```

```
Epoch[0], val_loss: 1.3355, val_acc: 0.5911
→ New best val_acc = 0.5911  (ckpt saved)
Epoch[1], val_loss: 1.4404, val_acc: 0.5958
→ New best val_acc = 0.5958  (ckpt saved)
Epoch[2], val_loss: 1.5870, val_acc: 0.5973
→ New best val_acc = 0.5973  (ckpt saved)
Epoch[3], val_loss: 1.7187, val_acc: 0.6013
→ New best val_acc = 0.6013  (ckpt saved)
Epoch[4], val_loss: 1.8676, val_acc: 0.6053
→ New best val_acc = 0.6053  (ckpt saved)
Epoch[5], val_loss: 2.0103, val_acc: 0.6048
Epoch[6], val_loss: 2.1499, val_acc: 0.6076
→ New best val_acc = 0.6076  (ckpt saved)
Epoch[7], val_loss: 2.2882, val_acc: 0.6051
Epoch[8], val_loss: 2.3629, val_acc: 0.6079
→ New best val_acc = 0.6079  (ckpt saved)
Epoch[9], val_loss: 2.4701, val_acc: 0.6068
Epoch[10], val_loss: 2.5653, val_acc: 0.6056
Epoch[11], val_loss: 2.6384, val_acc: 0.6076
Epoch[12], val_loss: 2.6702, val_acc: 0.6021
Epoch[13], val_loss: 2.7313, val_acc: 0.6024
Epoch[14], val_loss: 2.7850, val_acc: 0.6053
Epoch[15], val_loss: 2.8049, val_acc: 0.6033
Epoch[16], val_loss: 2.8382, val_acc: 0.6046
Epoch[17], val_loss: 2.8815, val_acc: 0.6010
Epoch[18], val_loss: 2.9082, val_acc: 0.6033
Epoch[19], val_loss: 2.9495, val_acc: 0.6046
Epoch[20], val_loss: 2.9367, val_acc: 0.6051
Epoch[21], val_loss: 2.9481, val_acc: 0.6064
Epoch[22], val_loss: 2.9492, val_acc: 0.6053
Epoch[23], val_loss: 2.9742, val_acc: 0.6028
Epoch[24], val_loss: 2.9983, val_acc: 0.6041
Epoch[25], val_loss: 3.0039, val_acc: 0.6051
Epoch[26], val_loss: 3.0030, val_acc: 0.6069
Epoch[27], val_loss: 3.0048, val_acc: 0.6046
Epoch[28], val_loss: 3.0340, val_acc: 0.6033
Early stopping at epoch 29 (no val_acc gain for 20 epochs).
```

Tuned 2

In [ ]:

```python
# Further fine tuning with reduced learning rates and epochs
model_eff_b0.load_state_dict(torch.load('model_efficientnetb0_tuned.pth'))

# ── Hyper-parameters for further fine-tuning ──────────────────────────
epochs_ft        = 100         # Reduced epochs for final tuning
batch_size_ft    = 16         # Further reduced batch size for better generalization
base_lr_ft       = 5e-5       # Further reduced base learning rate
max_lr_ft        = 5e-4       # Further reduced max learning rate
weight_decay_ft  = 0.025      # Further reduced weight decay
label_smooth_ft  = 0.025      # Further reduced label smoothing
grad_clip_norm   = 0.5        # Further reduced gradient clipping
patience_ft      = 25         # Increased patience for more stable training

# ❷  Optimiser with reduced learning rates
decay, no_decay = [], []
for n, p in model_eff_b0.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_ft = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_ft},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_ft, betas=(0.9, 0.999), eps=1e-8  # Higher beta2 for stability
)

# ❸  Learning rate schedule using RLROP
sched_ft = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_ft,
    mode='max',           # Monitor validation accuracy
    factor=0.2,          # More aggressive LR reduction
    patience=8,          # Increased patience before reducing LR
    verbose=True,        # Print message when LR changes
    min_lr=5e-8         # Lower minimum LR threshold
)

# ❺  Training loop with early-stop and checkpoint
best_val_acc_ft   = 0.0
epochs_no_imp_ft  = 0
history_ft        = []

for epoch in range(epochs_ft):
```

```
    # ---- TRAIN ------------------------------------------------------------
    model_eff_b0.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_b0.training_step(batch)
        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0.parameters(), grad_clip_norm)

        optimizer_ft.step()
        optimizer_ft.zero_grad(set_to_none=True)

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_b0, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0.epoch_end(epoch, result)
    history_ft.append(result)

    # Update learning rate based on validation accuracy
    sched_ft.step(result['val_acc'])

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_ft:
        best_val_acc_ft, epochs_no_imp_ft = val_acc, 0
        torch.save(model_eff_b0.state_dict(), 'model_efficientnetb0_tuned_2.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_ft += 1
        if epochs_no_imp_ft >= patience_ft:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_ft} epochs).")
            break
```

```
Epoch[0], val_loss: 2.4519, val_acc: 0.6066
→ New best val_acc = 0.6066  (ckpt saved)
Epoch[1], val_loss: 2.5080, val_acc: 0.6051
Epoch[2], val_loss: 2.5705, val_acc: 0.6054
Epoch[3], val_loss: 2.6014, val_acc: 0.6056
Epoch[4], val_loss: 2.6296, val_acc: 0.6064
Epoch[5], val_loss: 2.6666, val_acc: 0.6051
Epoch[6], val_loss: 2.7028, val_acc: 0.6066
→ New best val_acc = 0.6066  (ckpt saved)
Epoch[7], val_loss: 2.7527, val_acc: 0.6035
Epoch[8], val_loss: 2.7831, val_acc: 0.6058
Epoch[9], val_loss: 2.8155, val_acc: 0.6061
Epoch[10], val_loss: 2.8062, val_acc: 0.6063
Epoch[11], val_loss: 2.8207, val_acc: 0.6061
Epoch[12], val_loss: 2.8266, val_acc: 0.6073
→ New best val_acc = 0.6073  (ckpt saved)
Epoch[13], val_loss: 2.8340, val_acc: 0.6066
Epoch[14], val_loss: 2.8368, val_acc: 0.6058
Epoch[15], val_loss: 2.8336, val_acc: 0.6066
Epoch[16], val_loss: 2.8444, val_acc: 0.6069
Epoch[17], val_loss: 2.8626, val_acc: 0.6054
Epoch[18], val_loss: 2.8568, val_acc: 0.6069
Epoch[19], val_loss: 2.8721, val_acc: 0.6028
Epoch[20], val_loss: 2.8773, val_acc: 0.6063
Epoch[21], val_loss: 2.8648, val_acc: 0.6026
Epoch[22], val_loss: 2.8766, val_acc: 0.6063
Epoch[23], val_loss: 2.8766, val_acc: 0.6048
Epoch[24], val_loss: 2.8973, val_acc: 0.6048
Epoch[25], val_loss: 2.8723, val_acc: 0.6084
→ New best val_acc = 0.6084  (ckpt saved)
Epoch[26], val_loss: 2.8951, val_acc: 0.6046
Epoch[27], val_loss: 2.8857, val_acc: 0.6056
Epoch[28], val_loss: 2.8861, val_acc: 0.6068
Epoch[29], val_loss: 2.8977, val_acc: 0.6048
Epoch[30], val_loss: 2.9040, val_acc: 0.6038
Epoch[31], val_loss: 2.8911, val_acc: 0.6058
Epoch[32], val_loss: 2.8953, val_acc: 0.6058
Epoch[33], val_loss: 2.8931, val_acc: 0.6079
Epoch[34], val_loss: 2.8880, val_acc: 0.6046
Epoch[35], val_loss: 2.8860, val_acc: 0.6043
Epoch[36], val_loss: 2.8909, val_acc: 0.6058
Epoch[37], val_loss: 2.8876, val_acc: 0.6056
Epoch[38], val_loss: 2.8977, val_acc: 0.6066
Epoch[39], val_loss: 2.9052, val_acc: 0.6068
Epoch[40], val_loss: 2.8888, val_acc: 0.6068
Epoch[41], val_loss: 2.8866, val_acc: 0.6071
Epoch[42], val_loss: 2.9127, val_acc: 0.6038
Epoch[43], val_loss: 2.8997, val_acc: 0.6061
Epoch[44], val_loss: 2.9179, val_acc: 0.6076
Epoch[45], val_loss: 2.8931, val_acc: 0.6084
→ New best val_acc = 0.6084  (ckpt saved)
Epoch[46], val_loss: 2.8943, val_acc: 0.6051
Epoch[47], val_loss: 2.8986, val_acc: 0.6061
Epoch[48], val_loss: 2.8885, val_acc: 0.6063
Epoch[49], val_loss: 2.8995, val_acc: 0.6058
Epoch[50], val_loss: 2.8927, val_acc: 0.6066
Epoch[51], val_loss: 2.8884, val_acc: 0.6061
Epoch[52], val_loss: 2.9150, val_acc: 0.6053
Epoch[53], val_loss: 2.9032, val_acc: 0.6051
Epoch[54], val_loss: 2.9120, val_acc: 0.6061
Epoch[55], val_loss: 2.9002, val_acc: 0.6073
Epoch[56], val_loss: 2.9003, val_acc: 0.6046
Epoch[57], val_loss: 2.8958, val_acc: 0.6053
Epoch[58], val_loss: 2.8938, val_acc: 0.6053
Epoch[59], val_loss: 2.8991, val_acc: 0.6041
Epoch[60], val_loss: 2.8965, val_acc: 0.6066
Epoch[61], val_loss: 2.8971, val_acc: 0.6053
Epoch[62], val_loss: 2.8964, val_acc: 0.6076
Epoch[63], val_loss: 2.8994, val_acc: 0.6049
Epoch[64], val_loss: 2.8938, val_acc: 0.6041
Epoch[65], val_loss: 2.8921, val_acc: 0.6046
Epoch[66], val_loss: 2.9012, val_acc: 0.6066
Epoch[67], val_loss: 2.9048, val_acc: 0.6053
Epoch[68], val_loss: 2.8976, val_acc: 0.6051
Epoch[69], val_loss: 2.8940, val_acc: 0.6076
Epoch[70], val_loss: 2.8946, val_acc: 0.6081
Early stopping at epoch 71 (no val_acc gain for 25 epochs).
```

Tuned 3

In [ ]:

```python
# Further fine tuning with aggressive parameters
model_eff_b0.load_state_dict(torch.load('model_efficientnetb0_tuned_2.pth'))

# —— Hyper-parameters for aggressive fine-tuning ————————————————————————
```

```python
epochs_ft        = 200        # More epochs to push training further
batch_size_ft    = 128        # Larger batch size for more stable gradients
base_lr_ft       = 1e-4       # Higher base learning rate
max_lr_ft        = 1e-3       # Higher max learning rate
weight_decay_ft  = 0.03       # Increased weight decay for regularization
label_smooth_ft  = 0.15       # More label smoothing
grad_clip_norm   = 1.0        # Higher gradient clipping threshold
patience_ft      = 25         # More patience before early stopping

# ❷  Optimiser with weight decay separation
decay, no_decay = [], []
for n, p in model_eff_b0.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_ft = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_ft},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_ft, betas=(0.95, 0.999), eps=1e-8  # Higher momentum
)

# ❸  OneCycle learning rate scheduler
steps_per_epoch = len(train_dl)
total_steps = epochs_ft * steps_per_epoch

sched_ft = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_ft,
    max_lr=max_lr_ft,
    total_steps=total_steps,
    pct_start=0.35,          # Longer warmup
    div_factor=20,           # Higher initial learning rate
    final_div_factor=1e3,    # Higher final learning rate
    anneal_strategy='cos'    # Cosine annealing
)

# Define focal loss with class weights to handle imbalance
class_weights = 1.2 * torch.ones(7).to(device)  # Slightly higher class weights
focal_loss = torch.nn.CrossEntropyLoss(weight=class_weights, label_smoothing=label_smooth_ft, reduction='none')

def focal_loss_fn(outputs, targets, gamma=2.5):  # Higher gamma for more focus on hard examples
    ce_loss = focal_loss(outputs, targets)
    pt = torch.exp(-ce_loss)
    focal_weight = (1-pt)**gamma
    return (focal_weight * ce_loss).mean()

# ❻  Training loop with early-stop and checkpoint
best_val_acc_ft   = 0.0
epochs_no_imp_ft  = 0
history_ft        = []

for epoch in range(epochs_ft):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_b0.train()
    train_losses = []
    for batch in train_dl:
        images, labels = batch
        images, labels = images.to(device), labels.to(device)
        outputs = model_eff_b0(images)
        loss = focal_loss_fn(outputs, labels)
        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0.parameters(), grad_clip_norm)

        optimizer_ft.step()
        sched_ft.step()  # Step scheduler after each batch for OneCycleLR
        optimizer_ft.zero_grad(set_to_none=True)

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_b0, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0.epoch_end(epoch, result)
    history_ft.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_ft:
        best_val_acc_ft, epochs_no_imp_ft = val_acc, 0
        torch.save(model_eff_b0.state_dict(), 'model_efficientnetb0_tuned_3.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_ft += 1
        if epochs_no_imp_ft >= patience_ft:
```

```
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_ft} epochs).")
            break
```

```
Epoch[0], val_loss: 2.4806, val_acc: 0.5970
→ New best val_acc = 0.5970  (ckpt saved)
Epoch[1], val_loss: 2.0217, val_acc: 0.5862
Epoch[2], val_loss: 1.6648, val_acc: 0.5915
Epoch[3], val_loss: 1.4582, val_acc: 0.5953
Epoch[4], val_loss: 1.3507, val_acc: 0.5998
→ New best val_acc = 0.5998  (ckpt saved)
Epoch[5], val_loss: 1.3082, val_acc: 0.5961
Epoch[6], val_loss: 1.2811, val_acc: 0.6049
→ New best val_acc = 0.6049  (ckpt saved)
Epoch[7], val_loss: 1.2677, val_acc: 0.6074
→ New best val_acc = 0.6074  (ckpt saved)
Epoch[8], val_loss: 1.2714, val_acc: 0.6072
Epoch[9], val_loss: 1.2697, val_acc: 0.6072
Epoch[10], val_loss: 1.2747, val_acc: 0.6097
→ New best val_acc = 0.6097  (ckpt saved)
Epoch[11], val_loss: 1.2789, val_acc: 0.6074
Epoch[12], val_loss: 1.2908, val_acc: 0.6061
Epoch[13], val_loss: 1.2958, val_acc: 0.6079
Epoch[14], val_loss: 1.2968, val_acc: 0.6120
→ New best val_acc = 0.6120  (ckpt saved)
Epoch[15], val_loss: 1.3031, val_acc: 0.6117
Epoch[16], val_loss: 1.3322, val_acc: 0.6056
Epoch[17], val_loss: 1.3379, val_acc: 0.6081
Epoch[18], val_loss: 1.3248, val_acc: 0.6094
Epoch[19], val_loss: 1.3244, val_acc: 0.6109
Epoch[20], val_loss: 1.3467, val_acc: 0.6087
Epoch[21], val_loss: 1.3377, val_acc: 0.6083
Epoch[22], val_loss: 1.3506, val_acc: 0.6031
Epoch[23], val_loss: 1.3496, val_acc: 0.6094
Epoch[24], val_loss: 1.3427, val_acc: 0.6058
Epoch[25], val_loss: 1.3598, val_acc: 0.5995
Epoch[26], val_loss: 1.3639, val_acc: 0.6016
Epoch[27], val_loss: 1.3527, val_acc: 0.6069
Epoch[28], val_loss: 1.3579, val_acc: 0.6140
→ New best val_acc = 0.6140  (ckpt saved)
Epoch[29], val_loss: 1.3611, val_acc: 0.6072
Epoch[30], val_loss: 1.3676, val_acc: 0.5970
Epoch[31], val_loss: 1.3830, val_acc: 0.6013
Epoch[32], val_loss: 1.3449, val_acc: 0.6073
Epoch[33], val_loss: 1.3644, val_acc: 0.6015
Epoch[34], val_loss: 1.3557, val_acc: 0.6091
Epoch[35], val_loss: 1.3311, val_acc: 0.6116
Epoch[36], val_loss: 1.3492, val_acc: 0.6078
Epoch[37], val_loss: 1.3494, val_acc: 0.6046
Epoch[38], val_loss: 1.3618, val_acc: 0.6026
Epoch[39], val_loss: 1.3832, val_acc: 0.5998
Epoch[40], val_loss: 1.3781, val_acc: 0.6068
Epoch[41], val_loss: 1.3793, val_acc: 0.6003
Epoch[42], val_loss: 1.3632, val_acc: 0.6069
Epoch[43], val_loss: 1.3640, val_acc: 0.6117
Epoch[44], val_loss: 1.3399, val_acc: 0.6137
Epoch[45], val_loss: 1.3740, val_acc: 0.6081
Epoch[46], val_loss: 1.4166, val_acc: 0.6057
Epoch[47], val_loss: 1.4128, val_acc: 0.6007
Epoch[48], val_loss: 1.3629, val_acc: 0.6115
Epoch[49], val_loss: 1.3659, val_acc: 0.6064
Epoch[50], val_loss: 1.3446, val_acc: 0.6084
Epoch[51], val_loss: 1.3533, val_acc: 0.6049
Epoch[52], val_loss: 1.3967, val_acc: 0.5971
Epoch[53], val_loss: 1.3473, val_acc: 0.6079
Early stopping at epoch 54 (no val_acc gain for 25 epochs).
```

Tuned 4

In [ ]:

```
# Further fine tuning with aggressive parameters
model_eff_b0.load_state_dict(torch.load('model_efficientnetb0_tuned_3.pth'))

# —— Hyper-parameters for aggressive fine-tuning ——————————————————
epochs_ft        = 200        # More epochs to push training further
batch_size_ft    = 128        # Larger batch size for more stable gradients
base_lr_ft       = 1e-4       # Higher base learning rate
max_lr_ft        = 1e-3       # Higher max learning rate
weight_decay_ft  = 0.03       # Increased weight decay for regularization
label_smooth_ft  = 0.15       # More label smoothing
grad_clip_norm   = 1.0        # Higher gradient clipping threshold
patience_ft      = 25         # More patience before early stopping

# ②  Optimiser with weight decay separation
decay, no_decay = [], []
for n, p in model_eff_b0.named_parameters():
```

```
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_ft = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_ft},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_ft, betas=(0.95, 0.999), eps=1e-8  # Higher momentum
)

# 8  OneCycle learning rate scheduler
steps_per_epoch = len(train_dl)
total_steps = epochs_ft * steps_per_epoch

sched_ft = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_ft,
    max_lr=max_lr_ft,
    total_steps=total_steps,
    pct_start=0.35,          # Longer warmup
    div_factor=20,           # Higher initial learning rate
    final_div_factor=1e3,    # Higher final learning rate
    anneal_strategy='cos'    # Cosine annealing
)

# Define focal loss with class weights to handle imbalance
class_weights = 1.2 * torch.ones(7).to(device)  # Slightly higher class weights
focal_loss = torch.nn.CrossEntropyLoss(weight=class_weights, label_smoothing=label_smooth_ft, reduction='none')

def focal_loss_fn(outputs, targets, gamma=2.5):  # Higher gamma for more focus on hard examples
    ce_loss = focal_loss(outputs, targets)
    pt = torch.exp(-ce_loss)
    focal_weight = (1-pt)**gamma
    return (focal_weight * ce_loss).mean()

# 9  Training loop with early-stop and checkpoint
best_val_acc_ft   = 0.0
epochs_no_imp_ft  = 0
history_ft        = []

for epoch in range(epochs_ft):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_b0.train()
    train_losses = []
    for batch in train_dl:
        images, labels = batch
        images, labels = images.to(device), labels.to(device)
        outputs = model_eff_b0(images)
        loss = focal_loss_fn(outputs, labels)
        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0.parameters(), grad_clip_norm)

        optimizer_ft.step()
        sched_ft.step()  # Step scheduler after each batch for OneCycleLR
        optimizer_ft.zero_grad(set_to_none=True)

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_b0, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0.epoch_end(epoch, result)
    history_ft.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_ft:
        best_val_acc_ft, epochs_no_imp_ft = val_acc, 0
        torch.save(model_eff_b0.state_dict(), 'model_efficientnetb0_tuned_4.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_ft += 1
        if epochs_no_imp_ft >= patience_ft:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_ft} epochs).")
            break
```

```
Epoch[0], val_loss: 1.3328, val_acc: 0.6009
→ New best val_acc = 0.6009  (ckpt saved)
Epoch[1], val_loss: 1.2831, val_acc: 0.6054
→ New best val_acc = 0.6054  (ckpt saved)
Epoch[2], val_loss: 1.2679, val_acc: 0.6077
→ New best val_acc = 0.6077  (ckpt saved)
Epoch[3], val_loss: 1.2662, val_acc: 0.6084
→ New best val_acc = 0.6084  (ckpt saved)
Epoch[4], val_loss: 1.2684, val_acc: 0.6072
Epoch[5], val_loss: 1.2718, val_acc: 0.6082
Epoch[6], val_loss: 1.2758, val_acc: 0.6094
→ New best val_acc = 0.6094  (ckpt saved)
Epoch[7], val_loss: 1.2815, val_acc: 0.6074
Epoch[8], val_loss: 1.2931, val_acc: 0.6077
Epoch[9], val_loss: 1.2974, val_acc: 0.6109
→ New best val_acc = 0.6109  (ckpt saved)
Epoch[10], val_loss: 1.3105, val_acc: 0.6074
Epoch[11], val_loss: 1.3181, val_acc: 0.6112
→ New best val_acc = 0.6112  (ckpt saved)
Epoch[12], val_loss: 1.3248, val_acc: 0.6087
Epoch[13], val_loss: 1.3212, val_acc: 0.6089
Epoch[14], val_loss: 1.3409, val_acc: 0.6026
Epoch[15], val_loss: 1.3409, val_acc: 0.6053
Epoch[16], val_loss: 1.3416, val_acc: 0.6041
Epoch[17], val_loss: 1.3515, val_acc: 0.6053
Epoch[18], val_loss: 1.3316, val_acc: 0.6091
Epoch[19], val_loss: 1.3614, val_acc: 0.6047
Epoch[20], val_loss: 1.3547, val_acc: 0.6059
Epoch[21], val_loss: 1.3638, val_acc: 0.6051
Epoch[22], val_loss: 1.3503, val_acc: 0.6096
Epoch[23], val_loss: 1.3513, val_acc: 0.6086
Epoch[24], val_loss: 1.3666, val_acc: 0.5998
Epoch[25], val_loss: 1.3624, val_acc: 0.6033
Epoch[26], val_loss: 1.3791, val_acc: 0.6015
Epoch[27], val_loss: 1.3796, val_acc: 0.6064
Epoch[28], val_loss: 1.3655, val_acc: 0.6084
Epoch[29], val_loss: 1.3751, val_acc: 0.6016
Epoch[30], val_loss: 1.3645, val_acc: 0.6044
Epoch[31], val_loss: 1.3644, val_acc: 0.6061
Epoch[32], val_loss: 1.3711, val_acc: 0.6074
Epoch[33], val_loss: 1.3896, val_acc: 0.5995
Epoch[34], val_loss: 1.3802, val_acc: 0.5978
Epoch[35], val_loss: 1.3688, val_acc: 0.6081
Epoch[36], val_loss: 1.3728, val_acc: 0.6087
Early stopping at epoch 37 (no val_acc gain for 25 epochs).
```

In [ ]:

```python
from fvcore.nn import FlopCountAnalysis
# ———————— final test ————————————————————————————————————
# model_eff_b0.load_state_dict(torch.load('model_efficientnetb0.pth')) # 0.563
# model_eff_b0.load_state_dict(torch.load('model_efficientnetb0_tuned.pth')) # 0.616
# model_eff_b0.load_state_dict(torch.load('model_efficientnetb0_tuned_2.pth')) # 0.6177
# model_eff_b0.load_state_dict(torch.load('model_efficientnetb0_tuned_3.pth')) # 0.6205
model_eff_b0.load_state_dict(torch.load('model_efficientnetb0_tuned_4.pth')) # 0.6225



# 6) Final evaluation on the test set
result_eff_test = evaluate(model_eff_b0, test_dl)
print("EfficientNet-B0 Test result:", result_eff_test)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops_eff = FlopCountAnalysis(model_eff_b0.eval(), dummy_input)
gflops_eff = flops_eff.total() / 1e9
acc_eff = result_eff_test['val_acc']
efficiency_eff = acc_eff / gflops_eff
print(f"EfficientNet-B0 GFLOPs: {gflops_eff}")
print(f"EfficientNet-B0 Accuracy: {acc_eff}")
print(f"EfficientNet-B0 Efficiency: {efficiency_eff}")
```

```
Unsupported operator aten::silu_ encountered 49 time(s)
Unsupported operator aten::sigmoid encountered 16 time(s)
Unsupported operator aten::mul encountered 16 time(s)
Unsupported operator aten::add_ encountered 9 time(s)
Unsupported operator aten::dropout_ encountered 1 time(s)
The following submodules of the model were never called during the trace of the graph. They may be unused, or th
ey were accessed by direct calls to .forward() or via other python methods. In the latter case they will have ze
ros for statistics, though their statistics will still contribute to their parent calling module.
model.features.1.0.stochastic_depth, model.features.2.0.stochastic_depth, model.features.2.1.stochastic_depth, m
odel.features.3.0.stochastic_depth, model.features.3.1.stochastic_depth, model.features.4.0.stochastic_depth, mo
del.features.4.1.stochastic_depth, model.features.4.2.stochastic_depth, model.features.5.0.stochastic_depth, mod
el.features.5.1.stochastic_depth, model.features.5.2.stochastic_depth, model.features.6.0.stochastic_depth, mode
l.features.6.1.stochastic_depth, model.features.6.2.stochastic_depth, model.features.6.3.stochastic_depth, mode
l.features.7.0.stochastic_depth
```

```
EfficientNet-B0 Test result: {'val_loss': 1.275635004043579, 'val_acc': 0.6225019693374634}
EfficientNet-B0 GFLOPs: 0.023210688
EfficientNet-B0 Accuracy: 0.6225019693374634
EfficientNet-B0 Efficiency: 26.819625912746034
```

EfficientNet b0 trained on processed set (crop rotate flip)

Transform data

In [ ]:
```python
train_tsfm = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.RandomHorizontalFlip(p=0.5),
    T.RandomCrop(48, padding=4),
    T.RandomRotation(degrees=10),
    T.ToTensor(),
    T.Normalize(*stats,inplace=True),
])
valid_tsfm = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.ToTensor(),
    T.Normalize(*stats,inplace=True)
])

train_ds = expressions(train_df, train_tsfm)
valid_ds = expressions(valid_df, valid_tsfm)
test_ds = expressions(test_df, valid_tsfm)

batch_size = 400
train_dl = DataLoader(train_ds, batch_size, shuffle=True,
                      num_workers=0, pin_memory=False)
valid_dl = DataLoader(valid_ds, batch_size*2,
                    num_workers=0, pin_memory=False)
test_dl = DataLoader(test_ds, batch_size*2,
                    num_workers=0, pin_memory=False)

train_dl = DeviceDataLoader(train_dl, device)
valid_dl = DeviceDataLoader(valid_dl, device)
test_dl = DeviceDataLoader(test_dl, device)
```

Initial training

In [ ]:
```python
model_eff_b0_v2 = to_device(EfficientNetB0(7,1), device)
```

In [ ]:
```python
# ──────────────────────────────────────────────────────────────
#  EfficientNet-B0  →  optimised training loop (FER-2013)
# ──────────────────────────────────────────────────────────────
import math, torch

# ── Balanced Hyper-parameters ──────────────────────────────────
epochs_eff_b0_v2        = 300        # More epochs for training from scratch
batch_size_eff_b0_v2    = 64         # Larger batch size for smaller model
base_lr_eff_b0_v2       = 1e-3       # Higher LR since training from scratch
max_lr_eff_b0_v2        = 1e-2       # Higher max LR for full training
weight_decay_eff_b0_v2  = 0.1        # Standard weight decay
label_smooth_eff_b0_v2  = 0.1
grad_clip_norm       = 2.0        # Higher grad clip for from-scratch training
patience_eff_b0_v2      = 20         # Longer patience for full training

# ②  Optimiser — parameter-wise weight-decay split with layer-wise params
decay, no_decay = [], []
for n, p in model_eff_b0_v2.named_parameters():
    if not p.requires_grad: continue
    # Skip weight decay for biases, LayerNorm and position embeddings
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_eff_b0_v2 = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_eff_b0_v2},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_eff_b0_v2, betas=(0.9, 0.999), eps=1e-8  # Standard betas
)

# ③  Cosine scheduler with longer warmup for from-scratch training
sched_eff_b0_v2 = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_eff_b0_v2,
    max_lr          = max_lr_eff_b0_v2,
```

```
    epochs          = epochs_eff_b0_v2,
    steps_per_epoch = len(train_dl),
    pct_start       = 0.1,        # 10% warmup period for from-scratch
    div_factor      = 10,         # Standard LR range
    final_div_factor= 1000
)

# 6️  Training loop with early-stop and checkpoint
best_val_acc_eff_b0_v2   = 0.0
epochs_no_imp_eff_b0_v2  = 0
history_eff_b0_v2        = []

for epoch in range(epochs_eff_b0_v2):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_b0_v2.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_b0_v2.training_step(batch)       # CE + smooth inside
        train_losses.append(loss.detach())
        loss.backward()

        # clip gradients *before* stepping
        torch.nn.utils.clip_grad_norm_(model_eff_b0_v2.parameters(), grad_clip_norm)

        optimizer_eff_b0_v2.step()
        optimizer_eff_b0_v2.zero_grad(set_to_none=True)  # more efficient zero_grad
        sched_eff_b0_v2.step()

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_b0_v2, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0_v2.epoch_end(epoch, result)                  # prints nicely
    history_eff_b0_v2.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_eff_b0_v2:
        best_val_acc_eff_b0_v2, epochs_no_imp_eff_b0_v2 = val_acc, 0
        torch.save(model_eff_b0_v2.state_dict(), 'model_efficientnetb0_v2.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_eff_b0_v2 += 1
        if epochs_no_imp_eff_b0_v2 >= patience_eff_b0_v2:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_eff_b0_v2} epochs).")
            break
```

```
Epoch[0], val_loss: 1.6510, val_acc: 0.3617
→ New best val_acc = 0.3617  (ckpt saved)
Epoch[1], val_loss: 1.4560, val_acc: 0.4405
→ New best val_acc = 0.4405  (ckpt saved)
Epoch[2], val_loss: 1.3443, val_acc: 0.4923
→ New best val_acc = 0.4923  (ckpt saved)
Epoch[3], val_loss: 1.2492, val_acc: 0.5252
→ New best val_acc = 0.5252  (ckpt saved)
Epoch[4], val_loss: 1.1945, val_acc: 0.5507
→ New best val_acc = 0.5507  (ckpt saved)
Epoch[5], val_loss: 1.1703, val_acc: 0.5569
→ New best val_acc = 0.5569  (ckpt saved)
Epoch[6], val_loss: 1.1530, val_acc: 0.5668
→ New best val_acc = 0.5668  (ckpt saved)
Epoch[7], val_loss: 1.1529, val_acc: 0.5780
→ New best val_acc = 0.5780  (ckpt saved)
Epoch[8], val_loss: 1.1275, val_acc: 0.5816
→ New best val_acc = 0.5816  (ckpt saved)
Epoch[9], val_loss: 1.1194, val_acc: 0.5795
Epoch[10], val_loss: 1.1426, val_acc: 0.5897
→ New best val_acc = 0.5897  (ckpt saved)
Epoch[11], val_loss: 1.0991, val_acc: 0.5923
→ New best val_acc = 0.5923  (ckpt saved)
Epoch[12], val_loss: 1.1174, val_acc: 0.5929
→ New best val_acc = 0.5929  (ckpt saved)
Epoch[13], val_loss: 1.1152, val_acc: 0.5930
→ New best val_acc = 0.5930  (ckpt saved)
Epoch[14], val_loss: 1.0939, val_acc: 0.5928
Epoch[15], val_loss: 1.2129, val_acc: 0.5619
Epoch[16], val_loss: 1.1276, val_acc: 0.5874
Epoch[17], val_loss: 1.1009, val_acc: 0.5843
Epoch[18], val_loss: 1.1316, val_acc: 0.5927
Epoch[19], val_loss: 1.3350, val_acc: 0.5379
Epoch[20], val_loss: 1.1910, val_acc: 0.5486
Epoch[21], val_loss: 1.1261, val_acc: 0.5808
Epoch[22], val_loss: 1.1582, val_acc: 0.5626
Epoch[23], val_loss: 1.1914, val_acc: 0.5508
Epoch[24], val_loss: 1.1885, val_acc: 0.5598
Epoch[25], val_loss: 1.1417, val_acc: 0.5718
Epoch[26], val_loss: 1.2551, val_acc: 0.5308
Epoch[27], val_loss: 1.1663, val_acc: 0.5708
Epoch[28], val_loss: 1.1553, val_acc: 0.5842
Epoch[29], val_loss: 1.4227, val_acc: 0.4841
Epoch[30], val_loss: 1.0889, val_acc: 0.5851
Epoch[31], val_loss: 1.1227, val_acc: 0.5765
Epoch[32], val_loss: 1.1428, val_acc: 0.5663
Epoch[33], val_loss: 1.1145, val_acc: 0.5797
Early stopping at epoch 34 (no val_acc gain for 20 epochs).
```

Tune 1

In [ ]:

```python
# Further fine tuning with reduced learning rates and epochs
model_eff_b0_v2.load_state_dict(torch.load('model_efficientnetb0_v2.pth'))

# —— Hyper-parameters for further fine-tuning ————————————————————
epochs_ft        = 150         # One-third of initial epochs for fine-tuning
batch_size_ft    = 32         # Half the initial batch size for better generalization
base_lr_ft       = 1e-4       # One-tenth of initial base LR for fine-tuning
max_lr_ft        = 1e-3       # One-tenth of initial max LR for fine-tuning
weight_decay_ft  = 0.05       # Half the initial weight decay
label_smooth_ft  = 0.05       # Half the initial label smoothing
grad_clip_norm   = 1.0        # Half the initial gradient clipping
patience_ft      = 20         # Slightly reduced patience for fine-tuning

# ❷  Optimiser with reduced learning rates
decay, no_decay = [], []
for n, p in model_eff_b0_v2.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_ft = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_ft},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_ft, betas=(0.9, 0.99), eps=1e-8  # Higher beta2 for stability
)

# ❸  Learning rate schedule using RLROP
sched_ft = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_ft,
    mode='max',           # Monitor validation accuracy
    factor=0.5,          # Reduce LR by half when plateauing
```

```
    patience=5,          # Wait 5 epochs before reducing LR
    verbose=True,        # Print message when LR changes
    min_lr=1e-7         # Don't reduce LR below this
)

# 6  Training loop with early-stop and checkpoint
best_val_acc_ft   = 0.0
epochs_no_imp_ft  = 0
history_ft        = []

for epoch in range(epochs_ft):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_b0_v2.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_b0_v2.training_step(batch)
        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0_v2.parameters(), grad_clip_norm)

        optimizer_ft.step()
        optimizer_ft.zero_grad(set_to_none=True)

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_b0_v2, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0_v2.epoch_end(epoch, result)
    history_ft.append(result)

    # Update learning rate based on validation accuracy
    sched_ft.step(result['val_acc'])

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_ft:
        best_val_acc_ft, epochs_no_imp_ft = val_acc, 0
        torch.save(model_eff_b0_v2.state_dict(), 'model_efficientnetb0_v2_tuned.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_ft += 1
        if epochs_no_imp_ft >= patience_ft:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_ft} epochs).")
            break
```

```
Epoch[0], val_loss: 1.0120, val_acc: 0.6278
→ New best val_acc = 0.6278  (ckpt saved)
Epoch[1], val_loss: 1.0009, val_acc: 0.6351
→ New best val_acc = 0.6351  (ckpt saved)
Epoch[2], val_loss: 0.9946, val_acc: 0.6371
→ New best val_acc = 0.6371  (ckpt saved)
Epoch[3], val_loss: 0.9993, val_acc: 0.6382
→ New best val_acc = 0.6382  (ckpt saved)
Epoch[4], val_loss: 0.9977, val_acc: 0.6447
→ New best val_acc = 0.6447  (ckpt saved)
Epoch[5], val_loss: 0.9957, val_acc: 0.6386
Epoch[6], val_loss: 0.9958, val_acc: 0.6406
Epoch[7], val_loss: 0.9941, val_acc: 0.6394
Epoch[8], val_loss: 1.0029, val_acc: 0.6407
Epoch[9], val_loss: 1.0069, val_acc: 0.6384
Epoch[10], val_loss: 1.0068, val_acc: 0.6373
Epoch[11], val_loss: 1.0074, val_acc: 0.6429
Epoch[12], val_loss: 1.0116, val_acc: 0.6426
Epoch[13], val_loss: 1.0128, val_acc: 0.6449
→ New best val_acc = 0.6449  (ckpt saved)
Epoch[14], val_loss: 1.0142, val_acc: 0.6444
Epoch[15], val_loss: 1.0147, val_acc: 0.6432
Epoch[16], val_loss: 1.0202, val_acc: 0.6454
→ New best val_acc = 0.6454  (ckpt saved)
Epoch[17], val_loss: 1.0193, val_acc: 0.6421
Epoch[18], val_loss: 1.0219, val_acc: 0.6439
Epoch[19], val_loss: 1.0265, val_acc: 0.6467
→ New best val_acc = 0.6467  (ckpt saved)
Epoch[20], val_loss: 1.0308, val_acc: 0.6456
Epoch[21], val_loss: 1.0343, val_acc: 0.6492
→ New best val_acc = 0.6492  (ckpt saved)
Epoch[22], val_loss: 1.0381, val_acc: 0.6452
Epoch[23], val_loss: 1.0403, val_acc: 0.6459
Epoch[24], val_loss: 1.0393, val_acc: 0.6508
→ New best val_acc = 0.6508  (ckpt saved)
Epoch[25], val_loss: 1.0461, val_acc: 0.6479
Epoch[26], val_loss: 1.0457, val_acc: 0.6472
Epoch[27], val_loss: 1.0531, val_acc: 0.6454
Epoch[28], val_loss: 1.0550, val_acc: 0.6477
Epoch[29], val_loss: 1.0613, val_acc: 0.6466
Epoch[30], val_loss: 1.0612, val_acc: 0.6446
Epoch[31], val_loss: 1.0668, val_acc: 0.6475
Epoch[32], val_loss: 1.0682, val_acc: 0.6451
Epoch[33], val_loss: 1.0747, val_acc: 0.6454
Epoch[34], val_loss: 1.0758, val_acc: 0.6466
Epoch[35], val_loss: 1.0746, val_acc: 0.6494
Epoch[36], val_loss: 1.0778, val_acc: 0.6472
Epoch[37], val_loss: 1.0748, val_acc: 0.6490
Epoch[38], val_loss: 1.0829, val_acc: 0.6454
Epoch[39], val_loss: 1.0805, val_acc: 0.6459
Epoch[40], val_loss: 1.0818, val_acc: 0.6467
Epoch[41], val_loss: 1.0835, val_acc: 0.6472
Epoch[42], val_loss: 1.0872, val_acc: 0.6464
Epoch[43], val_loss: 1.0882, val_acc: 0.6494
Epoch[44], val_loss: 1.0877, val_acc: 0.6474
Early stopping at epoch 45 (no val_acc gain for 20 epochs).
```

Tune 2 with Label Smoothing Loss

In [ ]:

```python
# Further fine tuning with reduced learning rates and epochs
model_eff_b0_v2.load_state_dict(torch.load('model_efficientnetb0_v2_tuned.pth'))

# —— Hyper-parameters for further fine-tuning ——————————————————
epochs_ft        = 150         # One-third of initial epochs for fine-tuning
batch_size_ft    = 32         # Half the initial batch size for better generalization
base_lr_ft       = 5e-5       # One-tenth of initial base LR for fine-tuning
max_lr_ft        = 5e-4       # One-tenth of initial max LR for fine-tuning
weight_decay_ft  = 0.05       # Half the initial weight decay
label_smooth_ft  = 0.05       # Half the initial label smoothing
grad_clip_norm   = 1.0        # Half the initial gradient clipping
patience_ft      = 20         # Slightly reduced patience for fine-tuning

# Add Label Smoothing Loss
label_smooth_loss = torch.nn.CrossEntropyLoss(label_smoothing=label_smooth_ft)

# ② Optimiser with reduced learning rates
decay, no_decay = [], []
for n, p in model_eff_b0_v2.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)
```

```python
optimizer_ft = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_ft},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_ft, betas=(0.9, 0.99), eps=1e-8  # Higher beta2 for stability
)

# 8  Learning rate schedule using RLROP
sched_ft = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_ft,
    mode='max',           # Monitor validation accuracy
    factor=0.5,          # Reduce LR by half when plateauing
    patience=5,          # Wait 5 epochs before reducing LR
    verbose=True,        # Print message when LR changes
    min_lr=1e-7         # Don't reduce LR below this
)

# 9  Training loop with early-stop and checkpoint
best_val_acc_ft   = 0.0
epochs_no_imp_ft  = 0
history_ft        = []

for epoch in range(epochs_ft):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_b0_v2.train()
    train_losses = []
    for batch in train_dl:
        images, labels = batch
        images, labels = images.to(device), labels.to(device)

        # Forward pass
        outputs = model_eff_b0_v2(images)

        # Calculate loss with label smoothing
        loss = label_smooth_loss(outputs, labels)

        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0_v2.parameters(), grad_clip_norm)

        optimizer_ft.step()
        optimizer_ft.zero_grad(set_to_none=True)

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_b0_v2, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0_v2.epoch_end(epoch, result)
    history_ft.append(result)

    # Update learning rate based on validation accuracy
    sched_ft.step(result['val_acc'])

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_ft:
        best_val_acc_ft, epochs_no_imp_ft = val_acc, 0
        torch.save(model_eff_b0_v2.state_dict(), 'model_efficientnetb0_v2_tuned_2.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_ft += 1
        if epochs_no_imp_ft >= patience_ft:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_ft} epochs).")
            break
```

```
Epoch[0], val_loss: 1.0051, val_acc: 0.6472
→ New best val_acc = 0.6472  (ckpt saved)
Epoch[1], val_loss: 0.9963, val_acc: 0.6489
→ New best val_acc = 0.6489  (ckpt saved)
Epoch[2], val_loss: 0.9937, val_acc: 0.6517
→ New best val_acc = 0.6517  (ckpt saved)
Epoch[3], val_loss: 0.9930, val_acc: 0.6504
Epoch[4], val_loss: 0.9972, val_acc: 0.6496
Epoch[5], val_loss: 0.9996, val_acc: 0.6453
Epoch[6], val_loss: 1.0005, val_acc: 0.6504
Epoch[7], val_loss: 1.0024, val_acc: 0.6461
Epoch[8], val_loss: 1.0010, val_acc: 0.6436
Epoch[9], val_loss: 1.0033, val_acc: 0.6476
Epoch[10], val_loss: 1.0061, val_acc: 0.6474
Epoch[11], val_loss: 1.0089, val_acc: 0.6453
Epoch[12], val_loss: 1.0059, val_acc: 0.6484
Epoch[13], val_loss: 1.0080, val_acc: 0.6484
Epoch[14], val_loss: 1.0142, val_acc: 0.6492
Epoch[15], val_loss: 1.0136, val_acc: 0.6461
Epoch[16], val_loss: 1.0143, val_acc: 0.6469
Epoch[17], val_loss: 1.0147, val_acc: 0.6476
Epoch[18], val_loss: 1.0182, val_acc: 0.6474
Epoch[19], val_loss: 1.0190, val_acc: 0.6471
Epoch[20], val_loss: 1.0179, val_acc: 0.6461
Epoch[21], val_loss: 1.0193, val_acc: 0.6466
Epoch[22], val_loss: 1.0172, val_acc: 0.6474
Early stopping at epoch 23 (no val_acc gain for 20 epochs).
```

Test

In [ ]:
```python
from fvcore.nn import FlopCountAnalysis
# ———————— final test ————————————————————————————————
# model_eff_b0_v2.load_state_dict(torch.load('model_efficientnetb0_v2.pth')) # 0.5961
# model_eff_b0_v2.load_state_dict(torch.load('model_efficientnetb0_v2_tuned.pth')) # 0.6653
model_eff_b0_v2.load_state_dict(torch.load('model_efficientnetb0_v2_tuned_2.pth')) # 0.6666


# 6) Final evaluation on the test set
result_eff_test = evaluate(model_eff_b0_v2, test_dl)
print("EfficientNet-B0 Train Transform Test result:", result_eff_test)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops_eff = FlopCountAnalysis(model_eff_b0_v2.eval(), dummy_input)
gflops_eff = flops_eff.total() / 1e9
acc_eff = result_eff_test['val_acc']
efficiency_eff = acc_eff / gflops_eff
print(f"EfficientNet-B0 Train Transform GFLOPs: {gflops_eff}")
print(f"EfficientNet-B0 Train Transform Accuracy: {acc_eff}")
print(f"EfficientNet-B0 Train Transform Efficiency: {efficiency_eff}")
```

```
Unsupported operator aten::silu_ encountered 49 time(s)
Unsupported operator aten::sigmoid encountered 16 time(s)
Unsupported operator aten::mul encountered 16 time(s)
Unsupported operator aten::add_ encountered 9 time(s)
Unsupported operator aten::dropout_ encountered 1 time(s)
The following submodules of the model were never called during the trace of the graph. They may be unused, or th
ey were accessed by direct calls to .forward() or via other python methods. In the latter case they will have ze
ros for statistics, though their statistics will still contribute to their parent calling module.
model.features.1.0.stochastic_depth, model.features.2.0.stochastic_depth, model.features.2.1.stochastic_depth, m
odel.features.3.0.stochastic_depth, model.features.3.1.stochastic_depth, model.features.4.0.stochastic_depth, mo
del.features.4.1.stochastic_depth, model.features.4.2.stochastic_depth, model.features.5.0.stochastic_depth, mod
el.features.5.1.stochastic_depth, model.features.5.2.stochastic_depth, model.features.6.0.stochastic_depth, mode
l.features.6.1.stochastic_depth, model.features.6.2.stochastic_depth, model.features.6.3.stochastic_depth, mode
l.features.7.0.stochastic_depth
EfficientNet-B0 Train Transform Test result: {'val_loss': 0.9656729698181152, 'val_acc': 0.666577160358429}
EfficientNet-B0 Train Transform GFLOPs: 0.023210688
EfficientNet-B0 Train Transform Accuracy: 0.666577160358429
EfficientNet-B0 Train Transform Efficiency: 28.718543817332296
```

EfficientNetb0 trained on Random Brightness and Constrast

Transform data

In [ ]:
```python
train_tsfm = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.RandomHorizontalFlip(p=0.5),
    T.RandomCrop(48, padding=4),
    T.RandomRotation(degrees=10),
    T.ColorJitter(brightness=0.2, contrast=0.2),
```

```
    T.ToTensor(),
    T.Normalize(*stats,inplace=True),
])
valid_tsfm = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.ToTensor(),
    T.Normalize(*stats,inplace=True)
])

train_ds = expressions(train_df, train_tsfm)
valid_ds = expressions(valid_df, valid_tsfm)
test_ds = expressions(test_df, valid_tsfm)

batch_size = 400
train_dl = DataLoader(train_ds, batch_size, shuffle=True,
                      num_workers=0, pin_memory=False)
valid_dl = DataLoader(valid_ds, batch_size*2,
                    num_workers=0, pin_memory=False)
test_dl = DataLoader(test_ds, batch_size*2,
                     num_workers=0, pin_memory=False)

train_dl = DeviceDataLoader(train_dl, device)
valid_dl = DeviceDataLoader(valid_dl, device)
test_dl = DeviceDataLoader(test_dl, device)
```

Initial training

In [ ]:
```
model_eff_b0_v3 = to_device(EfficientNetB0(7,1), device)
```

In [ ]:
```
# ──────────────────────────────────────────────────────────────
#  EfficientNet-B0  →  optimised training loop (FER-2013)
# ──────────────────────────────────────────────────────────────
import math, torch

# ── Balanced Hyper-parameters ──────────────────────────────────
epochs_eff_b0_v3        = 300        # More epochs for training from scratch
batch_size_eff_b0_v3    = 64         # Larger batch size for smaller model
base_lr_eff_b0_v3       = 1e-3       # Higher LR since training from scratch
max_lr_eff_b0_v3        = 1e-2       # Higher max LR for full training
weight_decay_eff_b0_v3  = 0.1        # Standard weight decay
label_smooth_eff_b0_v3  = 0.1
grad_clip_norm       = 2.0        # Higher grad clip for from-scratch training
patience_eff_b0_v3      = 20         # Longer patience for full training

# ❷  Optimiser — parameter-wise weight-decay split with layer-wise params
decay, no_decay = [], []
for n, p in model_eff_b0_v3.named_parameters():
    if not p.requires_grad: continue
    # Skip weight decay for biases, LayerNorm and position embeddings
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_eff_b0_v3 = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_eff_b0_v3},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_eff_b0_v3, betas=(0.9, 0.999), eps=1e-8  # Standard betas
)

# ❸  Cosine scheduler with longer warmup for from-scratch training
sched_eff_b0_v3 = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_eff_b0_v3,
    max_lr          = max_lr_eff_b0_v3,
    epochs          = epochs_eff_b0_v3,
    steps_per_epoch = len(train_dl),
    pct_start       = 0.1,        # 10% warmup period for from-scratch
    div_factor      = 10,         # Standard LR range
    final_div_factor= 1000
)

# ❻  Training loop with early-stop and checkpoint
best_val_acc_eff_b0_v3   = 0.0
epochs_no_imp_eff_b0_v3  = 0
history_eff_b0_v3        = []

for epoch in range(epochs_eff_b0_v3):
    # ---- TRAIN ------------------------------------------------------
    model_eff_b0_v3.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_b0_v3.training_step(batch)       # CE + smooth inside
        train_losses.append(loss.detach())
```

```
        loss.backward()

        # clip gradients *before* stepping
        torch.nn.utils.clip_grad_norm_(model_eff_b0_v3.parameters(), grad_clip_norm)

        optimizer_eff_b0_v3.step()
        optimizer_eff_b0_v3.zero_grad(set_to_none=True)  # more efficient zero_grad
        sched_eff_b0_v3.step()

    # ---- VALIDATE ------------------------------------------------------
    result = evaluate(model_eff_b0_v3, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0_v3.epoch_end(epoch, result)                  # prints nicely
    history_eff_b0_v3.append(result)

    # ---- EARLY-STOP / CKPT ---------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_eff_b0_v3:
        best_val_acc_eff_b0_v3, epochs_no_imp_eff_b0_v3 = val_acc, 0
        torch.save(model_eff_b0_v3.state_dict(), 'model_efficientnetb0_v3.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_eff_b0_v3 += 1
        if epochs_no_imp_eff_b0_v3 >= patience_eff_b0_v3:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_eff_b0_v3} epochs).")
            break
```

```
Epoch[0], val_loss: 1.5888, val_acc: 0.3876
→ New best val_acc = 0.3876  (ckpt saved)
Epoch[1], val_loss: 1.3895, val_acc: 0.4673
→ New best val_acc = 0.4673  (ckpt saved)
Epoch[2], val_loss: 1.2772, val_acc: 0.5124
→ New best val_acc = 0.5124  (ckpt saved)
Epoch[3], val_loss: 1.2819, val_acc: 0.5206
→ New best val_acc = 0.5206  (ckpt saved)
Epoch[4], val_loss: 1.1972, val_acc: 0.5490
→ New best val_acc = 0.5490  (ckpt saved)
Epoch[5], val_loss: 1.1545, val_acc: 0.5628
→ New best val_acc = 0.5628  (ckpt saved)
Epoch[6], val_loss: 1.1685, val_acc: 0.5697
→ New best val_acc = 0.5697  (ckpt saved)
Epoch[7], val_loss: 1.1236, val_acc: 0.5792
→ New best val_acc = 0.5792  (ckpt saved)
Epoch[8], val_loss: 1.1620, val_acc: 0.5704
Epoch[9], val_loss: 1.1189, val_acc: 0.5825
→ New best val_acc = 0.5825  (ckpt saved)
Epoch[10], val_loss: 1.1348, val_acc: 0.5898
→ New best val_acc = 0.5898  (ckpt saved)
Epoch[11], val_loss: 1.1112, val_acc: 0.5851
Epoch[12], val_loss: 1.1245, val_acc: 0.5831
Epoch[13], val_loss: 1.1300, val_acc: 0.5877
Epoch[14], val_loss: 1.1574, val_acc: 0.5610
Epoch[15], val_loss: 1.1242, val_acc: 0.5840
Epoch[16], val_loss: 1.1255, val_acc: 0.5899
→ New best val_acc = 0.5899  (ckpt saved)
Epoch[17], val_loss: 1.1564, val_acc: 0.5732
Epoch[18], val_loss: 1.1686, val_acc: 0.5640
Epoch[19], val_loss: 1.1141, val_acc: 0.5833
Epoch[20], val_loss: 1.1416, val_acc: 0.5710
Epoch[21], val_loss: 1.1132, val_acc: 0.5912
→ New best val_acc = 0.5912  (ckpt saved)
Epoch[22], val_loss: 1.1412, val_acc: 0.5689
Epoch[23], val_loss: 1.0974, val_acc: 0.5806
Epoch[24], val_loss: 1.2070, val_acc: 0.5474
Epoch[25], val_loss: 1.1162, val_acc: 0.5878
Epoch[26], val_loss: 1.2368, val_acc: 0.5271
Epoch[27], val_loss: 1.1655, val_acc: 0.5445
Epoch[28], val_loss: 1.0944, val_acc: 0.5870
Epoch[29], val_loss: 1.0899, val_acc: 0.5999
→ New best val_acc = 0.5999  (ckpt saved)
Epoch[30], val_loss: 1.1813, val_acc: 0.5612
Epoch[31], val_loss: 1.0890, val_acc: 0.5868
Epoch[32], val_loss: 1.0882, val_acc: 0.5874
Epoch[33], val_loss: 1.1746, val_acc: 0.5609
Epoch[34], val_loss: 1.1528, val_acc: 0.5841
Epoch[35], val_loss: 1.1333, val_acc: 0.5765
Epoch[36], val_loss: 1.0750, val_acc: 0.6074
→ New best val_acc = 0.6074  (ckpt saved)
Epoch[37], val_loss: 1.0768, val_acc: 0.5929
Epoch[38], val_loss: 1.1682, val_acc: 0.5718
Epoch[39], val_loss: 1.1899, val_acc: 0.5535
Epoch[40], val_loss: 1.1514, val_acc: 0.5731
Epoch[41], val_loss: 1.0956, val_acc: 0.5892
Epoch[42], val_loss: 1.1535, val_acc: 0.5639
Epoch[43], val_loss: 1.1189, val_acc: 0.5888
Epoch[44], val_loss: 1.0940, val_acc: 0.5969
Epoch[45], val_loss: 1.1118, val_acc: 0.5848
Epoch[46], val_loss: 1.0961, val_acc: 0.5971
Epoch[47], val_loss: 1.1554, val_acc: 0.5817
Epoch[48], val_loss: 1.2098, val_acc: 0.5582
Epoch[49], val_loss: 1.1050, val_acc: 0.5805
Epoch[50], val_loss: 1.1315, val_acc: 0.5886
Epoch[51], val_loss: 1.1201, val_acc: 0.5810
Epoch[52], val_loss: 1.1258, val_acc: 0.5918
Epoch[53], val_loss: 1.1350, val_acc: 0.5828
Epoch[54], val_loss: 1.1580, val_acc: 0.5890
Epoch[55], val_loss: 1.2554, val_acc: 0.5624
Epoch[56], val_loss: 1.0871, val_acc: 0.5882
Early stopping at epoch 57 (no val_acc gain for 20 epochs).
```

Tune 1

In [ ]:

```python
# Further fine tuning with reduced learning rates and epochs
model_eff_b0_v3.load_state_dict(torch.load('model_efficientnetb0_v3.pth'))

# —— Hyper-parameters for further fine-tuning ——————————————————————
epochs_ft        = 150         # One-third of initial epochs for fine-tuning
batch_size_ft    = 32         # Half the initial batch size for better generalization
base_lr_ft       = 5e-5       # One-tenth of initial base LR for fine-tuning
max_lr_ft        = 5e-4       # One-tenth of initial max LR for fine-tuning
weight_decay_ft  = 0.05       # Half the initial weight decay
```

```python
label_smooth_ft  = 0.05       # Half the initial label smoothing
grad_clip_norm   = 1.0        # Half the initial gradient clipping
patience_ft      = 20         # Slightly reduced patience for fine-tuning

# ②  Optimiser with reduced learning rates
decay, no_decay = [], []
for n, p in model_eff_b0_v3.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_ft = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_ft},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_ft, betas=(0.9, 0.99), eps=1e-8  # Higher beta2 for stability
)

# ③  Learning rate schedule using RLROP
sched_ft = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_ft,
    mode='max',           # Monitor validation accuracy
    factor=0.5,          # Reduce LR by half when plateauing
    patience=5,          # Wait 5 epochs before reducing LR
    verbose=True,        # Print message when LR changes
    min_lr=1e-7         # Don't reduce LR below this
)

# ④  Training loop with early-stop and checkpoint
best_val_acc_ft   = 0.0
epochs_no_imp_ft  = 0
history_ft        = []

for epoch in range(epochs_ft):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_b0_v3.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_b0_v3.training_step(batch)
        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0_v3.parameters(), grad_clip_norm)

        optimizer_ft.step()
        optimizer_ft.zero_grad(set_to_none=True)

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_b0_v3, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0_v3.epoch_end(epoch, result)
    history_ft.append(result)

    # Update learning rate based on validation accuracy
    sched_ft.step(result['val_acc'])

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_ft:
        best_val_acc_ft, epochs_no_imp_ft = val_acc, 0
        torch.save(model_eff_b0_v3.state_dict(), 'model_efficientnetb0_v3_tuned.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_ft += 1
        if epochs_no_imp_ft >= patience_ft:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_ft} epochs).")
            break
```

```
Epoch[0], val_loss: 1.1484, val_acc: 0.6032
→ New best val_acc = 0.6032  (ckpt saved)
Epoch[1], val_loss: 1.0858, val_acc: 0.6106
→ New best val_acc = 0.6106  (ckpt saved)
Epoch[2], val_loss: 1.0624, val_acc: 0.6127
→ New best val_acc = 0.6127  (ckpt saved)
Epoch[3], val_loss: 1.0496, val_acc: 0.6187
→ New best val_acc = 0.6187  (ckpt saved)
Epoch[4], val_loss: 1.0427, val_acc: 0.6220
→ New best val_acc = 0.6220  (ckpt saved)
Epoch[5], val_loss: 1.0403, val_acc: 0.6207
Epoch[6], val_loss: 1.0350, val_acc: 0.6230
→ New best val_acc = 0.6230  (ckpt saved)
Epoch[7], val_loss: 1.0329, val_acc: 0.6252
→ New best val_acc = 0.6252  (ckpt saved)
Epoch[8], val_loss: 1.0318, val_acc: 0.6248
Epoch[9], val_loss: 1.0303, val_acc: 0.6262
→ New best val_acc = 0.6262  (ckpt saved)
Epoch[10], val_loss: 1.0253, val_acc: 0.6283
→ New best val_acc = 0.6283  (ckpt saved)
Epoch[11], val_loss: 1.0252, val_acc: 0.6295
→ New best val_acc = 0.6295  (ckpt saved)
Epoch[12], val_loss: 1.0200, val_acc: 0.6313
→ New best val_acc = 0.6313  (ckpt saved)
Epoch[13], val_loss: 1.0209, val_acc: 0.6291
Epoch[14], val_loss: 1.0195, val_acc: 0.6313
Epoch[15], val_loss: 1.0185, val_acc: 0.6323
→ New best val_acc = 0.6323  (ckpt saved)
Epoch[16], val_loss: 1.0212, val_acc: 0.6311
Epoch[17], val_loss: 1.0180, val_acc: 0.6290
Epoch[18], val_loss: 1.0147, val_acc: 0.6318
Epoch[19], val_loss: 1.0155, val_acc: 0.6306
Epoch[20], val_loss: 1.0129, val_acc: 0.6296
Epoch[21], val_loss: 1.0158, val_acc: 0.6293
Epoch[22], val_loss: 1.0175, val_acc: 0.6308
Epoch[23], val_loss: 1.0173, val_acc: 0.6306
Epoch[24], val_loss: 1.0178, val_acc: 0.6306
Epoch[25], val_loss: 1.0186, val_acc: 0.6318
Epoch[26], val_loss: 1.0201, val_acc: 0.6338
→ New best val_acc = 0.6338  (ckpt saved)
Epoch[27], val_loss: 1.0207, val_acc: 0.6333
Epoch[28], val_loss: 1.0186, val_acc: 0.6320
Epoch[29], val_loss: 1.0174, val_acc: 0.6318
Epoch[30], val_loss: 1.0196, val_acc: 0.6333
Epoch[31], val_loss: 1.0198, val_acc: 0.6326
Epoch[32], val_loss: 1.0205, val_acc: 0.6323
Epoch[33], val_loss: 1.0206, val_acc: 0.6323
Epoch[34], val_loss: 1.0184, val_acc: 0.6343
→ New best val_acc = 0.6343  (ckpt saved)
Epoch[35], val_loss: 1.0210, val_acc: 0.6328
Epoch[36], val_loss: 1.0209, val_acc: 0.6346
→ New best val_acc = 0.6346  (ckpt saved)
Epoch[37], val_loss: 1.0188, val_acc: 0.6353
→ New best val_acc = 0.6353  (ckpt saved)
Epoch[38], val_loss: 1.0222, val_acc: 0.6356
→ New best val_acc = 0.6356  (ckpt saved)
Epoch[39], val_loss: 1.0230, val_acc: 0.6351
Epoch[40], val_loss: 1.0226, val_acc: 0.6316
Epoch[41], val_loss: 1.0224, val_acc: 0.6346
Epoch[42], val_loss: 1.0230, val_acc: 0.6341
Epoch[43], val_loss: 1.0201, val_acc: 0.6341
Epoch[44], val_loss: 1.0235, val_acc: 0.6353
Epoch[45], val_loss: 1.0224, val_acc: 0.6356
Epoch[46], val_loss: 1.0239, val_acc: 0.6343
Epoch[47], val_loss: 1.0230, val_acc: 0.6356
Epoch[48], val_loss: 1.0237, val_acc: 0.6379
→ New best val_acc = 0.6379  (ckpt saved)
Epoch[49], val_loss: 1.0242, val_acc: 0.6346
Epoch[50], val_loss: 1.0261, val_acc: 0.6341
Epoch[51], val_loss: 1.0233, val_acc: 0.6353
Epoch[52], val_loss: 1.0211, val_acc: 0.6358
Epoch[53], val_loss: 1.0229, val_acc: 0.6346
Epoch[54], val_loss: 1.0228, val_acc: 0.6351
Epoch[55], val_loss: 1.0250, val_acc: 0.6353
Epoch[56], val_loss: 1.0232, val_acc: 0.6356
Epoch[57], val_loss: 1.0217, val_acc: 0.6353
Epoch[58], val_loss: 1.0239, val_acc: 0.6361
Epoch[59], val_loss: 1.0254, val_acc: 0.6351
Epoch[60], val_loss: 1.0251, val_acc: 0.6349
Epoch[61], val_loss: 1.0242, val_acc: 0.6343
Epoch[62], val_loss: 1.0245, val_acc: 0.6348
Epoch[63], val_loss: 1.0227, val_acc: 0.6341
Epoch[64], val_loss: 1.0238, val_acc: 0.6356
Epoch[65], val_loss: 1.0261, val_acc: 0.6356
Epoch[66], val_loss: 1.0227, val_acc: 0.6348
Epoch[67], val_loss: 1.0248, val_acc: 0.6358
```

```
Epoch[68], val_loss: 1.0231, val_acc: 0.6336
Early stopping at epoch 69 (no val_acc gain for 20 epochs).
```

Test

In [ ]:
```python
from fvcore.nn import FlopCountAnalysis
# ———————— final test ——————————————————————————————————————
# model_eff_b0_v3.load_state_dict(torch.load('model_efficientnetb0_v3.pth')) # 0.58
model_eff_b0_v3.load_state_dict(torch.load('model_efficientnetb0_v3_tuned.pth')) # 0.6451

# 6) Final evaluation on the test set
result_eff_test = evaluate(model_eff_b0_v3, test_dl)
print("EfficientNet-B0 Train Transform Test result:", result_eff_test)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops_eff = FlopCountAnalysis(model_eff_b0_v3.eval(), dummy_input)
gflops_eff = flops_eff.total() / 1e9
acc_eff = result_eff_test['val_acc']
efficiency_eff = acc_eff / gflops_eff
print(f"EfficientNet-B0 Train Transform GFLOPs: {gflops_eff}")
print(f"EfficientNet-B0 Train Transform Accuracy: {acc_eff}")
print(f"EfficientNet-B0 Train Transform Efficiency: {efficiency_eff}")
```

```
Unsupported operator aten::silu_ encountered 49 time(s)
Unsupported operator aten::sigmoid encountered 16 time(s)
Unsupported operator aten::mul encountered 16 time(s)
Unsupported operator aten::add_ encountered 9 time(s)
Unsupported operator aten::dropout_ encountered 1 time(s)
The following submodules of the model were never called during the trace of the graph. They may be unused, or th
ey were accessed by direct calls to .forward() or via other python methods. In the latter case they will have ze
ros for statistics, though their statistics will still contribute to their parent calling module.
model.features.1.0.stochastic_depth, model.features.2.0.stochastic_depth, model.features.2.1.stochastic_depth, m
odel.features.3.0.stochastic_depth, model.features.3.1.stochastic_depth, model.features.4.0.stochastic_depth, mo
del.features.4.1.stochastic_depth, model.features.4.2.stochastic_depth, model.features.5.0.stochastic_depth, mod
el.features.5.1.stochastic_depth, model.features.5.2.stochastic_depth, model.features.6.0.stochastic_depth, mode
l.features.6.1.stochastic_depth, model.features.6.2.stochastic_depth, model.features.6.3.stochastic_depth, mode
l.features.7.0.stochastic_depth
EfficientNet-B0 Train Transform Test result: {'val_loss': 0.9971514940261841, 'val_acc': 0.6451432704925537}
EfficientNet-B0 Train Transform GFLOPs: 0.023210688
EfficientNet-B0 Train Transform Accuracy: 0.6451432704925537
EfficientNet-B0 Train Transform Efficiency: 27.795094677613765
```

EfficientNetb0 trained on processed and Random Cut Out

Transform data

In [ ]:
```python
train_tsfm = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.RandomHorizontalFlip(p=0.5),
    T.RandomCrop(48, padding=4),
    T.RandomRotation(degrees=10),
    T.ToTensor(),
    T.Normalize(*stats,inplace=True),
    T.RandomErasing(p=0.5, scale=(0.01, 0.11), ratio=(1.0, 1.0), value=0),
])

valid_tsfm = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.ToTensor(),
    T.Normalize(*stats,inplace=True)
])

train_ds = expressions(train_df, train_tsfm)
valid_ds = expressions(valid_df, valid_tsfm)
test_ds = expressions(test_df, valid_tsfm)

batch_size = 400
train_dl = DataLoader(train_ds, batch_size, shuffle=True,
                      num_workers=0, pin_memory=False)
valid_dl = DataLoader(valid_ds, batch_size*2,
                    num_workers=0, pin_memory=False)
test_dl = DataLoader(test_ds, batch_size*2,
                    num_workers=0, pin_memory=False)

train_dl = DeviceDataLoader(train_dl, device)
valid_dl = DeviceDataLoader(valid_dl, device)
test_dl = DeviceDataLoader(test_dl, device)
```

Initial training

In [ ]:
```
model_eff_b0_v4 = to_device(EfficientNetB0(7,1), device)
```

In [ ]:
```
# ──────────────────────────────────────────────────────────────────────
#  EfficientNet-B0  →  optimised training loop (FER-2013)
# ──────────────────────────────────────────────────────────────────────
import math, torch

# ── Balanced Hyper-parameters ──────────────────────────────────────────
epochs_eff_b0_v4        = 300        # More epochs for training from scratch
batch_size_eff_b0_v4    = 64         # Larger batch size for smaller model
base_lr_eff_b0_v4       = 1e-3       # Higher LR since training from scratch
max_lr_eff_b0_v4        = 1e-2       # Higher max LR for full training
weight_decay_eff_b0_v4  = 0.1        # Standard weight decay
label_smooth_eff_b0_v4  = 0.1
grad_clip_norm       = 2.0        # Higher grad clip for from-scratch training
patience_eff_b0_v4      = 20         # Longer patience for full training

# ❷  Optimiser — parameter-wise weight-decay split with layer-wise params
decay, no_decay = [], []
for n, p in model_eff_b0_v4.named_parameters():
    if not p.requires_grad: continue
    # Skip weight decay for biases, LayerNorm and position embeddings
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_eff_b0_v4 = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_eff_b0_v4},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_eff_b0_v4, betas=(0.9, 0.999), eps=1e-8  # Standard betas
)

# ❸  Cosine scheduler with longer warmup for from-scratch training
sched_eff_b0_v4 = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_eff_b0_v4,
    max_lr          = max_lr_eff_b0_v4,
    epochs          = epochs_eff_b0_v4,
    steps_per_epoch = len(train_dl),
    pct_start       = 0.1,        # 10% warmup period for from-scratch
    div_factor      = 10,         # Standard LR range
    final_div_factor= 1000
)

# ❻  Training loop with early-stop and checkpoint
best_val_acc_eff_b0_v4   = 0.0
epochs_no_imp_eff_b0_v4  = 0
history_eff_b0_v4        = []

for epoch in range(epochs_eff_b0_v4):
    # ---- TRAIN ------------------------------------------------------
    model_eff_b0_v4.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_b0_v4.training_step(batch)       # CE + smooth inside
        train_losses.append(loss.detach())
        loss.backward()

        # clip gradients *before* stepping
        torch.nn.utils.clip_grad_norm_(model_eff_b0_v4.parameters(), grad_clip_norm)

        optimizer_eff_b0_v4.step()
        optimizer_eff_b0_v4.zero_grad(set_to_none=True)  # more efficient zero_grad
        sched_eff_b0_v4.step()

    # ---- VALIDATE ---------------------------------------------------
    result = evaluate(model_eff_b0_v4, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0_v4.epoch_end(epoch, result)                  # prints nicely
    history_eff_b0_v4.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_eff_b0_v4:
        best_val_acc_eff_b0_v4, epochs_no_imp_eff_b0_v4 = val_acc, 0
        torch.save(model_eff_b0_v4.state_dict(), 'model_efficientnetb0_v4.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_eff_b0_v4 += 1
        if epochs_no_imp_eff_b0_v4 >= patience_eff_b0_v4:
            print(f"Early stopping at epoch {epoch+1} "
```

```
                f"(no val_acc gain for {patience_eff_b0_v4} epochs).")
            break
```

```
Epoch[0], val_loss: 1.6300, val_acc: 0.3654
→ New best val_acc = 0.3654  (ckpt saved)
Epoch[1], val_loss: 1.4079, val_acc: 0.4669
→ New best val_acc = 0.4669  (ckpt saved)
Epoch[2], val_loss: 1.3042, val_acc: 0.4969
→ New best val_acc = 0.4969  (ckpt saved)
Epoch[3], val_loss: 1.2237, val_acc: 0.5304
→ New best val_acc = 0.5304  (ckpt saved)
Epoch[4], val_loss: 1.1972, val_acc: 0.5446
→ New best val_acc = 0.5446  (ckpt saved)
Epoch[5], val_loss: 1.1605, val_acc: 0.5589
→ New best val_acc = 0.5589  (ckpt saved)
Epoch[6], val_loss: 1.1474, val_acc: 0.5639
→ New best val_acc = 0.5639  (ckpt saved)
Epoch[7], val_loss: 1.1869, val_acc: 0.5691
→ New best val_acc = 0.5691  (ckpt saved)
Epoch[8], val_loss: 1.1374, val_acc: 0.5837
→ New best val_acc = 0.5837  (ckpt saved)
Epoch[9], val_loss: 1.0967, val_acc: 0.5886
→ New best val_acc = 0.5886  (ckpt saved)
Epoch[10], val_loss: 1.1476, val_acc: 0.5667
Epoch[11], val_loss: 1.0855, val_acc: 0.5826
Epoch[12], val_loss: 1.1025, val_acc: 0.5933
→ New best val_acc = 0.5933  (ckpt saved)
Epoch[13], val_loss: 1.0699, val_acc: 0.6025
→ New best val_acc = 0.6025  (ckpt saved)
Epoch[14], val_loss: 1.1811, val_acc: 0.5686
Epoch[15], val_loss: 1.1476, val_acc: 0.5782
Epoch[16], val_loss: 1.1689, val_acc: 0.5558
Epoch[17], val_loss: 1.2110, val_acc: 0.5430
Epoch[18], val_loss: 1.1187, val_acc: 0.5868
Epoch[19], val_loss: 1.1479, val_acc: 0.5755
Epoch[20], val_loss: 1.1084, val_acc: 0.5806
Epoch[21], val_loss: 1.1134, val_acc: 0.5832
Epoch[22], val_loss: 1.1363, val_acc: 0.5722
Epoch[23], val_loss: 1.1293, val_acc: 0.5634
Epoch[24], val_loss: 1.1039, val_acc: 0.5919
Epoch[25], val_loss: 1.1396, val_acc: 0.5749
Epoch[26], val_loss: 1.2309, val_acc: 0.5371
Epoch[27], val_loss: 1.1474, val_acc: 0.5781
Epoch[28], val_loss: 1.2181, val_acc: 0.5491
Epoch[29], val_loss: 1.1479, val_acc: 0.5616
Epoch[30], val_loss: 1.1129, val_acc: 0.5928
Epoch[31], val_loss: 1.0950, val_acc: 0.5920
Epoch[32], val_loss: 1.1569, val_acc: 0.5734
Epoch[33], val_loss: 1.1797, val_acc: 0.5636
Early stopping at epoch 34 (no val_acc gain for 20 epochs).
```

Tune 1

In [ ]:

```
# Further fine tuning with reduced learning rates and epochs
model_eff_b0_v4.load_state_dict(torch.load('model_efficientnetb0_v4.pth'))

# —— Hyper-parameters for further fine-tuning ————————————————————
epochs_ft        = 150         # One-third of initial epochs for fine-tuning
batch_size_ft    = 32         # Half the initial batch size for better generalization
base_lr_ft       = 5e-5       # One-tenth of initial base LR for fine-tuning
max_lr_ft        = 5e-4       # One-tenth of initial max LR for fine-tuning
weight_decay_ft  = 0.05       # Half the initial weight decay
label_smooth_ft  = 0.05       # Half the initial label smoothing
grad_clip_norm   = 1.0        # Half the initial gradient clipping
patience_ft      = 20         # Slightly reduced patience for fine-tuning

# ❷  Optimiser with reduced learning rates
decay, no_decay = [], []
for n, p in model_eff_b0_v4.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_ft = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_ft},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_ft, betas=(0.9, 0.99), eps=1e-8  # Higher beta2 for stability
)

# ❸  Learning rate schedule using RLROP
sched_ft = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_ft,
    mode='max',           # Monitor validation accuracy
```

```
    factor=0.5,          # Reduce LR by half when plateauing
    patience=5,          # Wait 5 epochs before reducing LR
    verbose=True,        # Print message when LR changes
    min_lr=1e-7         # Don't reduce LR below this
)

# 6  Training loop with early-stop and checkpoint
best_val_acc_ft   = 0.0
epochs_no_imp_ft  = 0
history_ft        = []

for epoch in range(epochs_ft):
    # ---- TRAIN ------------------------------------------------------------
    model_eff_b0_v4.train()
    train_losses = []
    for batch in train_dl:
        loss = model_eff_b0_v4.training_step(batch)
        train_losses.append(loss.detach())
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0_v4.parameters(), grad_clip_norm)

        optimizer_ft.step()
        optimizer_ft.zero_grad(set_to_none=True)

    # ---- VALIDATE ---------------------------------------------------------
    result = evaluate(model_eff_b0_v4, valid_dl)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0_v4.epoch_end(epoch, result)
    history_ft.append(result)

    # Update learning rate based on validation accuracy
    sched_ft.step(result['val_acc'])

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_ft:
        best_val_acc_ft, epochs_no_imp_ft = val_acc, 0
        torch.save(model_eff_b0_v4.state_dict(), 'model_efficientnetb0_v4_tuned.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
    else:
        epochs_no_imp_ft += 1
        if epochs_no_imp_ft >= patience_ft:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_ft} epochs).")
            break
```

```
Epoch[0], val_loss: 1.0152, val_acc: 0.6158
→ New best val_acc = 0.6158  (ckpt saved)
Epoch[1], val_loss: 1.0020, val_acc: 0.6239
→ New best val_acc = 0.6239  (ckpt saved)
Epoch[2], val_loss: 0.9940, val_acc: 0.6278
→ New best val_acc = 0.6278  (ckpt saved)
Epoch[3], val_loss: 0.9881, val_acc: 0.6280
→ New best val_acc = 0.6280  (ckpt saved)
Epoch[4], val_loss: 0.9832, val_acc: 0.6315
→ New best val_acc = 0.6315  (ckpt saved)
Epoch[5], val_loss: 0.9802, val_acc: 0.6345
→ New best val_acc = 0.6345  (ckpt saved)
Epoch[6], val_loss: 0.9781, val_acc: 0.6337
Epoch[7], val_loss: 0.9756, val_acc: 0.6365
→ New best val_acc = 0.6365  (ckpt saved)
Epoch[8], val_loss: 0.9728, val_acc: 0.6378
→ New best val_acc = 0.6378  (ckpt saved)
Epoch[9], val_loss: 0.9733, val_acc: 0.6395
→ New best val_acc = 0.6395  (ckpt saved)
Epoch[10], val_loss: 0.9714, val_acc: 0.6418
→ New best val_acc = 0.6418  (ckpt saved)
Epoch[11], val_loss: 0.9703, val_acc: 0.6418
Epoch[12], val_loss: 0.9677, val_acc: 0.6423
→ New best val_acc = 0.6423  (ckpt saved)
Epoch[13], val_loss: 0.9674, val_acc: 0.6393
Epoch[14], val_loss: 0.9682, val_acc: 0.6466
→ New best val_acc = 0.6466  (ckpt saved)
Epoch[15], val_loss: 0.9690, val_acc: 0.6450
Epoch[16], val_loss: 0.9671, val_acc: 0.6460
Epoch[17], val_loss: 0.9669, val_acc: 0.6451
Epoch[18], val_loss: 0.9676, val_acc: 0.6443
Epoch[19], val_loss: 0.9673, val_acc: 0.6455
Epoch[20], val_loss: 0.9685, val_acc: 0.6458
Epoch[21], val_loss: 0.9670, val_acc: 0.6468
→ New best val_acc = 0.6468  (ckpt saved)
Epoch[22], val_loss: 0.9671, val_acc: 0.6491
→ New best val_acc = 0.6491  (ckpt saved)
Epoch[23], val_loss: 0.9673, val_acc: 0.6473
Epoch[24], val_loss: 0.9678, val_acc: 0.6468
Epoch[25], val_loss: 0.9678, val_acc: 0.6468
Epoch[26], val_loss: 0.9676, val_acc: 0.6463
Epoch[27], val_loss: 0.9668, val_acc: 0.6488
Epoch[28], val_loss: 0.9676, val_acc: 0.6506
→ New best val_acc = 0.6506  (ckpt saved)
Epoch[29], val_loss: 0.9681, val_acc: 0.6465
Epoch[30], val_loss: 0.9668, val_acc: 0.6483
Epoch[31], val_loss: 0.9681, val_acc: 0.6455
Epoch[32], val_loss: 0.9700, val_acc: 0.6473
Epoch[33], val_loss: 0.9677, val_acc: 0.6453
Epoch[34], val_loss: 0.9698, val_acc: 0.6470
Epoch[35], val_loss: 0.9703, val_acc: 0.6463
Epoch[36], val_loss: 0.9696, val_acc: 0.6475
Epoch[37], val_loss: 0.9662, val_acc: 0.6491
Epoch[38], val_loss: 0.9685, val_acc: 0.6458
Epoch[39], val_loss: 0.9681, val_acc: 0.6463
Epoch[40], val_loss: 0.9702, val_acc: 0.6468
Epoch[41], val_loss: 0.9692, val_acc: 0.6488
Epoch[42], val_loss: 0.9692, val_acc: 0.6493
Epoch[43], val_loss: 0.9694, val_acc: 0.6488
Epoch[44], val_loss: 0.9712, val_acc: 0.6478
Epoch[45], val_loss: 0.9702, val_acc: 0.6493
Epoch[46], val_loss: 0.9714, val_acc: 0.6468
Epoch[47], val_loss: 0.9705, val_acc: 0.6455
Epoch[48], val_loss: 0.9722, val_acc: 0.6480
Early stopping at epoch 49 (no val_acc gain for 20 epochs).
```

Test

In [ ]:

```python
from fvcore.nn import FlopCountAnalysis
# ———————— final test ————————————————————————————————————
# model_eff_b0_v4.load_state_dict(torch.load('model_efficientnetb0_v4.pth')) # 0.6033
model_eff_b0_v4.load_state_dict(torch.load('model_efficientnetb0_v4_tuned.pth')) # 0.6522

# 6) Final evaluation on the test set
result_eff_test = evaluate(model_eff_b0_v4, test_dl)
print("EfficientNet-B0 Train Transform Test result:", result_eff_test)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops_eff = FlopCountAnalysis(model_eff_b0_v4.eval(), dummy_input)
gflops_eff = flops_eff.total() / 1e9
acc_eff = result_eff_test['val_acc']
efficiency_eff = acc_eff / gflops_eff
print(f"EfficientNet-B0 Train Transform GFLOPs: {gflops_eff}")
```

```
print(f"EfficientNet-B0 Train Transform Accuracy: {acc_eff}")
print(f"EfficientNet-B0 Train Transform Efficiency: {efficiency_eff}")
```

```
Unsupported operator aten::silu_ encountered 49 time(s)
Unsupported operator aten::sigmoid encountered 16 time(s)
Unsupported operator aten::mul encountered 16 time(s)
Unsupported operator aten::add_ encountered 9 time(s)
Unsupported operator aten::dropout_ encountered 1 time(s)
The following submodules of the model were never called during the trace of the graph. They may be unused, or th
ey were accessed by direct calls to .forward() or via other python methods. In the latter case they will have ze
ros for statistics, though their statistics will still contribute to their parent calling module.
model.features.1.0.stochastic_depth, model.features.2.0.stochastic_depth, model.features.2.1.stochastic_depth, m
odel.features.3.0.stochastic_depth, model.features.3.1.stochastic_depth, model.features.4.0.stochastic_depth, mo
del.features.4.1.stochastic_depth, model.features.4.2.stochastic_depth, model.features.5.0.stochastic_depth, mod
el.features.5.1.stochastic_depth, model.features.5.2.stochastic_depth, model.features.6.0.stochastic_depth, mode
l.features.6.1.stochastic_depth, model.features.6.2.stochastic_depth, model.features.6.3.stochastic_depth, mode
l.features.7.0.stochastic_depth
EfficientNet-B0 Train Transform Test result: {'val_loss': 0.955763041973114, 'val_acc': 0.6521574258804321}
EfficientNet-B0 Train Transform GFLOPs: 0.023210688
EfficientNet-B0 Train Transform Accuracy: 0.6521574258804321
EfficientNet-B0 Train Transform Efficiency: 28.097289743433375
```

EfficientNetb0 trained on MixUp CutMix

Transform data

In [ ]:

```
import torchvision.transforms as T
from torch.utils.data import DataLoader

# --- HYPERPARAMETERS for MixUp/CutMix ---
USE_MIXUP = True  # Set to True to use MixUp
MIXUP_ALPHA = 0.4 # Alpha parameter for Beta distribution in MixUp (e.g., 0.2, 0.4)

USE_CUTMIX = False # Set to True to use CutMix (usually not both MixUp and CutMix at the same time)
CUTMIX_ALPHA = 1.0 # Alpha parameter for Beta distribution in CutMix (often 1.0)
CUTMIX_PROB = 0.5 # Probability of applying CutMix per batch if USE_CUTMIX is True

# --- New Transforms for the MixUp/CutMix Model ---
# Keep base augmentations simpler as MixUp/CutMix are the primary regularizers
train_tsfm_mixup_cutmix = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.RandomHorizontalFlip(p=0.5), # Keep this, it's generally good
    # Remove RandomCrop, RandomRotation, RandomErasing for this specific model
    # as MixUp/CutMix will provide strong regularization.
    T.Resize((48, 48)), # Ensure images are the correct size if not cropping
    T.ToTensor(),
    T.Normalize(*stats, inplace=True),
])

# Validation and Test transforms remain the same
valid_tsfm = T.Compose([
    T.ToPILImage(),
    T.Grayscale(num_output_channels=1),
    T.Resize((48, 48)), # Ensure consistent size
    T.ToTensor(),
    T.Normalize(*stats, inplace=True)
])

# --- Create Datasets and DataLoaders for the MixUp/CutMix Model ---
train_ds_mc = expressions(train_df, train_tsfm_mixup_cutmix)
valid_ds_mc = expressions(valid_df, valid_tsfm) # Use original valid_tsfm
test_ds_mc = expressions(test_df, valid_tsfm)   # Use original valid_tsfm

batch_size = 64 # You might need to adjust batch size based on memory with MixUp/CutMix
                # Your original code had batch_size_eff_b0_v4 = 64 for initial training

train_dl_mc = DataLoader(train_ds_mc, batch_size, shuffle=True,
                         num_workers=0, pin_memory=False) # Adjust num_workers if needed
valid_dl_mc = DataLoader(valid_ds_mc, batch_size*2, # Use your preferred batch size for validation
                         num_workers=0, pin_memory=False)
test_dl_mc = DataLoader(test_ds_mc, batch_size*2,
                        num_workers=0, pin_memory=False)

# Assuming DeviceDataLoader and device are defined elsewhere
train_dl_mc = DeviceDataLoader(train_dl_mc, device)
valid_dl_mc = DeviceDataLoader(valid_dl_mc, device)
test_dl_mc = DeviceDataLoader(test_dl_mc, device)

print("DataLoaders for MixUp/CutMix model created.")
print(f"Using MixUp: {USE_MIXUP}, Alpha: {MIXUP_ALPHA if USE_MIXUP else 'N/A'}")
print(f"Using CutMix: {USE_CUTMIX}, Alpha: {CUTMIX_ALPHA if USE_CUTMIX else 'N/A'}, Prob: {CUTMIX_PROB if USE_C
```

```
DataLoaders for MixUp/CutMix model created.
Using MixUp: True, Alpha: 0.4
Using CutMix: False, Alpha: N/A, Prob: N/A
```

In [ ]:
```python
import numpy as np
import torch

def mixup_data(x, y, alpha=1.0, device='cuda'):
    '''Returns mixed inputs, pairs of targets, and lambda'''
    if alpha > 0:
        lam = np.random.beta(alpha, alpha)
    else:
        lam = 1

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

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

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

def rand_bbox(size, lam):
    W = size[2]
    H = size[3]
    cut_rat = np.sqrt(1. - lam)
    cut_w = int(W * cut_rat)
    cut_h = int(H * cut_rat)

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

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

    return bbx1, bby1, bbx2, bby2

def cutmix_data(x, y, alpha=1.0, device='cuda'):
    '''Returns mixed inputs, pairs of targets, and lambda'''
    if alpha > 0:
        lam = np.random.beta(alpha, alpha)
    else:
        lam = 1

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

    y_a, y_b = y, y[index]
    bbx1, bby1, bbx2, bby2 = rand_bbox(x.size(), lam)
    x_mixed = x.clone() # Clone to avoid modifying the original tensor in place if it's passed around
    x_mixed[:, :, bbx1:bbx2, bby1:bby2] = x[index, :, bbx1:bbx2, bby1:bby2]
    # adjust lambda to exactly match pixel ratio
    lam = 1 - ((bbx2 - bbx1) * (bby2 - bby1) / (x.size()[-1] * x.size()[-2]))
    return x_mixed, y_a, y_b, lam

# Use the same mixup_criterion for CutMix as the logic is the same for weighted loss
cutmix_criterion = mixup_criterion
```

Initial training

In [ ]:
```python
model_eff_b0_v5 = to_device(EfficientNetB0(7,1), device)
```

In [ ]:
```python
# ──────────────────────────────────────────────────────────────
#  EfficientNet-B0 V5 (MixUp/CutMix) →  optimised training loop
# ──────────────────────────────────────────────────────────────

# ── Balanced Hyper-parameters ──────────────────────────────────
epochs_eff_b0_v5        = 300
batch_size_eff_b0_v5    = 64 # This is the batch_size for the DataLoaders
base_lr_eff_b0_v5       = 1e-3
max_lr_eff_b0_v5        = 1e-2
weight_decay_eff_b0_v5  = 0.1
label_smooth_eff_b0_v5  = 0.1 # This will be used by the base criterion
grad_clip_norm          = 2.0
patience_eff_b0_v5      = 20

# Create the base criterion with label smoothing for this model
base_criterion_v5 = nn.CrossEntropyLoss(label_smoothing=label_smooth_eff_b0_v5)
print(f"Base criterion for v5: {base_criterion_v5}")
```

```python
# ❷  Optimiser
decay, no_decay = [], []
for n, p in model_eff_b0_v5.named_parameters():
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
        no_decay.append(p)
    else:
        decay.append(p)

optimizer_eff_b0_v5 = torch.optim.AdamW(
    [{'params': decay,    'weight_decay': weight_decay_eff_b0_v5},
     {'params': no_decay, 'weight_decay': 0.0}],
    lr=base_lr_eff_b0_v5, betas=(0.9, 0.999), eps=1e-8
)

# ❸  Scheduler
# IMPORTANT: Use train_dl_mc for steps_per_epoch
sched_eff_b0_v5 = torch.optim.lr_scheduler.OneCycleLR(
    optimizer_eff_b0_v5,
    max_lr          = max_lr_eff_b0_v5,
    epochs          = epochs_eff_b0_v5,
    steps_per_epoch = len(train_dl_mc), # Use the correct DataLoader
    pct_start       = 0.1,
    div_factor      = 10,
    final_div_factor= 1000
)

# ❹  Training loop with early-stop and checkpoint
best_val_acc_eff_b0_v5   = 0.0
epochs_no_imp_eff_b0_v5  = 0
history_eff_b0_v5        = []

print(f"\nStarting training for EfficientNet-B0 v5 (MixUp/CutMix)...")
print(f"Using train_dl_mc with length: {len(train_dl_mc)}")
print(f"Using valid_dl_mc with length: {len(valid_dl_mc)}")


for epoch in range(epochs_eff_b0_v5):
    model_eff_b0_v5.train()
    train_losses = []
    for batch_data in train_dl_mc: # Use the MixUp/CutMix DataLoader
        images, labels = batch_data # Already on device due to DeviceDataLoader
        # images, labels = images.to(device), labels.to(device) # Redundant if using DeviceDataLoader

        optimizer_eff_b0_v5.zero_grad(set_to_none=True)

        loss = 0
        # Determine if CutMix should be applied this batch
        apply_cutmix_this_batch = USE_CUTMIX and not USE_MIXUP and np.random.rand() < CUTMIX_PROB
        # If both USE_MIXUP and USE_CUTMIX are true, this logic prioritizes MixUp.
        # Adjust if you want to randomly pick between them.

        if USE_MIXUP:
            mixed_images, labels_a, labels_b, lam = mixup_data(images, labels, MIXUP_ALPHA, device)
            outputs = model_eff_b0_v5(mixed_images)
            loss = mixup_criterion(base_criterion_v5, outputs, labels_a, labels_b, lam)
        elif apply_cutmix_this_batch: # Only if USE_MIXUP is False and USE_CUTMIX is True
            mixed_images, labels_a, labels_b, lam = cutmix_data(images, labels, CUTMIX_ALPHA, device)
            outputs = model_eff_b0_v5(mixed_images)
            loss = cutmix_criterion(base_criterion_v5, outputs, labels_a, labels_b, lam)
        else: # Standard training (if both USE_MIXUP and USE_CUTMIX are False)
            outputs = model_eff_b0_v5(images)
            loss = base_criterion_v5(outputs, labels)

        train_losses.append(loss.detach()) # Detach before appending
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0_v5.parameters(), grad_clip_norm)
        optimizer_eff_b0_v5.step()
        sched_eff_b0_v5.step() # OneCycleLR steps after each batch

    # ---- VALIDATE ----
    # Use the correct validation DataLoader: valid_dl_mc
    result = evaluate(model_eff_b0_v5, valid_dl_mc)
    result['train_loss'] = torch.stack(train_losses).mean().item()
    model_eff_b0_v5.epoch_end(epoch, result)
    history_eff_b0_v5.append(result)

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc = result['val_acc']
    if val_acc > best_val_acc_eff_b0_v5:
        best_val_acc_eff_b0_v5, epochs_no_imp_eff_b0_v5 = val_acc, 0
        torch.save(model_eff_b0_v5.state_dict(), 'model_efficientnetb0_v5.pth')
        print(f"→ New best val_acc = {val_acc:.4f}  (ckpt saved)")
```

```
    else:
        epochs_no_imp_eff_b0_v5 += 1
        if epochs_no_imp_eff_b0_v5 >= patience_eff_b0_v5:
            print(f"Early stopping at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_eff_b0_v5} epochs).")
            break

print("Training for v5 finished.")
```

```
Base criterion for v5: CrossEntropyLoss()

Starting training for EfficientNet-B0 v5 (MixUp/CutMix)...
Using train_dl_mc with length: 449
Using valid_dl_mc with length: 29
Epoch[0], val_loss: 1.4794, val_acc: 0.4501
→ New best val_acc = 0.4501  (ckpt saved)
Epoch[1], val_loss: 1.2913, val_acc: 0.5208
→ New best val_acc = 0.5208  (ckpt saved)
Epoch[2], val_loss: 1.2730, val_acc: 0.5310
→ New best val_acc = 0.5310  (ckpt saved)
Epoch[3], val_loss: 1.2146, val_acc: 0.5542
→ New best val_acc = 0.5542  (ckpt saved)
Epoch[4], val_loss: 1.1879, val_acc: 0.5821
→ New best val_acc = 0.5821  (ckpt saved)
Epoch[5], val_loss: 1.1966, val_acc: 0.5674
Epoch[6], val_loss: 1.2042, val_acc: 0.5673
Epoch[7], val_loss: 1.2194, val_acc: 0.5711
Epoch[8], val_loss: 1.1748, val_acc: 0.5955
→ New best val_acc = 0.5955  (ckpt saved)
Epoch[9], val_loss: 1.2468, val_acc: 0.5520
Epoch[10], val_loss: 1.2185, val_acc: 0.5495
Epoch[11], val_loss: 1.2947, val_acc: 0.5182
Epoch[12], val_loss: 1.3977, val_acc: 0.4860
Epoch[13], val_loss: 1.2896, val_acc: 0.5446
Epoch[14], val_loss: 1.2735, val_acc: 0.5263
Epoch[15], val_loss: 1.3299, val_acc: 0.5177
Epoch[16], val_loss: 1.3989, val_acc: 0.4618
Epoch[17], val_loss: 1.3096, val_acc: 0.5267
Epoch[18], val_loss: 1.3140, val_acc: 0.5288
Epoch[19], val_loss: 1.4626, val_acc: 0.4697
Epoch[20], val_loss: 1.3514, val_acc: 0.5057
Epoch[21], val_loss: 1.3568, val_acc: 0.5091
Epoch[22], val_loss: 1.4141, val_acc: 0.4887
Epoch[23], val_loss: 1.3756, val_acc: 0.5032
Epoch[24], val_loss: 1.5734, val_acc: 0.4307
Epoch[25], val_loss: 1.3982, val_acc: 0.4862
Epoch[26], val_loss: 1.3828, val_acc: 0.4665
Epoch[27], val_loss: 1.3287, val_acc: 0.5041
Epoch[28], val_loss: 1.4134, val_acc: 0.5040
Early stopping at epoch 29 (no val_acc gain for 20 epochs).
Training for v5 finished.
```

Tune 1

In [ ]:

```
model_eff_b0_v5.load_state_dict(torch.load('model_efficientnetb0_v5.pth'))
print("\nLoaded model_efficientnetb0_v5.pth for fine-tuning.")

# —— Hyper-parameters for further fine-tuning v5 ——————————————————
epochs_ft_v5        = 150         # One-third of initial epochs for fine-tuning
batch_size_ft_v5    = 32          # Half the initial batch size for better generalization
                                  # Note: The DataLoaders (train_dl_mc, valid_dl_mc) were created with batch_si
                                  # If you want a different batch size for fine-tuning, you need to recreate
                                  # train_dl_mc_ft and valid_dl_mc_ft with this new batch_size_ft_v5.
                                  # For simplicity here, we'll assume you continue with the batch size used for
                                  # or adjust the DataLoaders accordingly if batch_size_ft_v5 is critical.
                                  # If keeping original batch size:
                                  # batch_size_ft_v5 = batch_size_eff_b0_v5 # Keep batch size from initial trai

base_lr_ft_v5       = 5e-5        # One-tenth of initial base LR for fine-tuning
# max_lr_ft_v5 is not used with ReduceLROnPlateau, but good to define if switching schedulers
weight_decay_ft_v5  = 0.05        # Half the initial weight decay
label_smooth_ft_v5  = 0.05        # Half the initial label smoothing for the base criterion
grad_clip_norm_ft_v5 = 1.0         # Half the initial gradient clipping
patience_ft_v5      = 20          # Slightly reduced patience for fine-tuning

# Create the base criterion for fine-tuning with the new label smoothing
base_criterion_ft_v5 = nn.CrossEntropyLoss(label_smoothing=label_smooth_ft_v5)
print(f"Base criterion for v5 fine-tuning: {base_criterion_ft_v5}")


# 🔒  Optimiser with reduced learning rates for v5 fine-tuning
decay_ft, no_decay_ft = [], []
for n, p in model_eff_b0_v5.named_parameters(): # Use model_eff_b0_v5
    if not p.requires_grad: continue
    if p.dim()==1 or n.endswith(".bias") or "layer_norm" in n or "position" in n:
```

```
        no_decay_ft.append(p)
    else:
        decay_ft.append(p)

optimizer_ft_v5 = torch.optim.AdamW(
    [{'params': decay_ft,    'weight_decay': weight_decay_ft_v5},
     {'params': no_decay_ft, 'weight_decay': 0.0}],
    lr=base_lr_ft_v5, betas=(0.9, 0.99), eps=1e-8
)

# 8️⃣  Learning rate schedule using RLROP for v5 fine-tuning
sched_ft_v5 = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_ft_v5,
    mode='max',
    factor=0.5,
    patience=5, # RLROP patience, not to be confused with early stopping patience_ft_v5
    verbose=True,
    min_lr=1e-7
)

# 9️⃣  Fine-tuning loop with early-stop and checkpoint for v5
best_val_acc_ft_v5   = 0.0
epochs_no_imp_ft_v5  = 0
history_ft_v5        = []

print(f"\nStarting fine-tuning for EfficientNet-B0 v5 (MixUp/CutMix)...")
# Ensure you use the correct dataloaders (train_dl_mc / valid_dl_mc or new _ft versions)
print(f"Using train_dl_mc (length: {len(train_dl_mc)}) for fine-tuning.")
print(f"Using valid_dl_mc (length: {len(valid_dl_mc)}) for fine-tuning validation.")

for epoch in range(epochs_ft_v5):
    model_eff_b0_v5.train()
    train_losses_ft = []
    for batch_data in train_dl_mc: # Use the MixUp/CutMix DataLoader (or train_dl_mc_ft if batch size changed)
        images, labels = batch_data

        optimizer_ft_v5.zero_grad(set_to_none=True)

        loss_ft = 0
        apply_cutmix_this_batch_ft = USE_CUTMIX and not USE_MIXUP and np.random.rand() < CUTMIX_PROB

        if USE_MIXUP:
            mixed_images, labels_a, labels_b, lam = mixup_data(images, labels, MIXUP_ALPHA, device)
            outputs = model_eff_b0_v5(mixed_images)
            # Use the fine-tuning base criterion
            loss_ft = mixup_criterion(base_criterion_ft_v5, outputs, labels_a, labels_b, lam)
        elif apply_cutmix_this_batch_ft:
            mixed_images, labels_a, labels_b, lam = cutmix_data(images, labels, CUTMIX_ALPHA, device)
            outputs = model_eff_b0_v5(mixed_images)
            # Use the fine-tuning base criterion
            loss_ft = cutmix_criterion(base_criterion_ft_v5, outputs, labels_a, labels_b, lam)
        else: # Standard training (if both USE_MIXUP and USE_CUTMIX are False)
            outputs = model_eff_b0_v5(images)
            # Use the fine-tuning base criterion
            loss_ft = base_criterion_ft_v5(outputs, labels)

        train_losses_ft.append(loss_ft.detach())
        loss_ft.backward()

        torch.nn.utils.clip_grad_norm_(model_eff_b0_v5.parameters(), grad_clip_norm_ft_v5)
        optimizer_ft_v5.step()
        # Note: ReduceLROnPlateau scheduler steps per epoch, after validation

    # ---- VALIDATE ----
    # Use the correct validation DataLoader: valid_dl_mc (or valid_dl_mc_ft)
    result_ft = evaluate(model_eff_b0_v5, valid_dl_mc) # Or valid_dl_mc_ft

    result_ft['train_loss'] = torch.stack(train_losses_ft).mean().item()
    model_eff_b0_v5.epoch_end(epoch, result_ft) # Assuming epoch_end is generic
    history_ft_v5.append(result_ft)

    # Update learning rate based on validation accuracy
    sched_ft_v5.step(result_ft['val_acc']) # Step RLROP scheduler

    # ---- EARLY-STOP / CKPT ------------------------------------------------
    val_acc_ft = result_ft['val_acc']
    if val_acc_ft > best_val_acc_ft_v5:
        best_val_acc_ft_v5, epochs_no_imp_ft_v5 = val_acc_ft, 0
        torch.save(model_eff_b0_v5.state_dict(), 'model_efficientnetb0_v5_tuned.pth') # Save as v5_tuned
        print(f"→ New best val_acc during v5 fine-tuning = {val_acc_ft:.4f}  (ckpt saved as model_efficientnetb
    else:
        epochs_no_imp_ft_v5 += 1
        if epochs_no_imp_ft_v5 >= patience_ft_v5:
            print(f"Early stopping during v5 fine-tuning at epoch {epoch+1} "
                  f"(no val_acc gain for {patience_ft_v5} epochs).")
```

```
            break
print("Fine-tuning for v5 finished.")
```

```
Loaded model_efficientnetb0_v5.pth for fine-tuning.
Base criterion for v5 fine-tuning: CrossEntropyLoss()

Starting fine-tuning for EfficientNet-B0 v5 (MixUp/CutMix)...
Using train_dl_mc (length: 449) for fine-tuning.
Using valid_dl_mc (length: 29) for fine-tuning validation.
Epoch[0], val_loss: 1.0862, val_acc: 0.6126
→ New best val_acc during v5 fine-tuning = 0.6126  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[1], val_loss: 1.0809, val_acc: 0.6182
→ New best val_acc during v5 fine-tuning = 0.6182  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[2], val_loss: 1.0593, val_acc: 0.6228
→ New best val_acc during v5 fine-tuning = 0.6228  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[3], val_loss: 1.0583, val_acc: 0.6249
→ New best val_acc during v5 fine-tuning = 0.6249  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[4], val_loss: 1.0538, val_acc: 0.6249
Epoch[5], val_loss: 1.0471, val_acc: 0.6276
→ New best val_acc during v5 fine-tuning = 0.6276  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[6], val_loss: 1.0452, val_acc: 0.6295
→ New best val_acc during v5 fine-tuning = 0.6295  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[7], val_loss: 1.0297, val_acc: 0.6314
→ New best val_acc during v5 fine-tuning = 0.6314  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[8], val_loss: 1.0311, val_acc: 0.6328
→ New best val_acc during v5 fine-tuning = 0.6328  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[9], val_loss: 1.0297, val_acc: 0.6328
Epoch[10], val_loss: 1.0391, val_acc: 0.6346
→ New best val_acc during v5 fine-tuning = 0.6346  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[11], val_loss: 1.0563, val_acc: 0.6248
Epoch[12], val_loss: 1.0254, val_acc: 0.6363
→ New best val_acc during v5 fine-tuning = 0.6363  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[13], val_loss: 1.0452, val_acc: 0.6346
Epoch[14], val_loss: 1.0378, val_acc: 0.6371
→ New best val_acc during v5 fine-tuning = 0.6371  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[15], val_loss: 1.0248, val_acc: 0.6360
Epoch[16], val_loss: 1.0221, val_acc: 0.6392
→ New best val_acc during v5 fine-tuning = 0.6392  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[17], val_loss: 1.0254, val_acc: 0.6390
Epoch[18], val_loss: 1.0234, val_acc: 0.6381
Epoch[19], val_loss: 1.0211, val_acc: 0.6400
→ New best val_acc during v5 fine-tuning = 0.6400  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[20], val_loss: 1.0243, val_acc: 0.6365
Epoch[21], val_loss: 1.0177, val_acc: 0.6400
Epoch[22], val_loss: 1.0203, val_acc: 0.6411
→ New best val_acc during v5 fine-tuning = 0.6411  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[23], val_loss: 1.0329, val_acc: 0.6381
Epoch[24], val_loss: 1.0362, val_acc: 0.6406
Epoch[25], val_loss: 1.0233, val_acc: 0.6395
Epoch[26], val_loss: 1.0085, val_acc: 0.6443
→ New best val_acc during v5 fine-tuning = 0.6443  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[27], val_loss: 1.0368, val_acc: 0.6398
Epoch[28], val_loss: 1.0261, val_acc: 0.6443
Epoch[29], val_loss: 1.0533, val_acc: 0.6307
Epoch[30], val_loss: 1.0356, val_acc: 0.6353
Epoch[31], val_loss: 1.0397, val_acc: 0.6422
Epoch[32], val_loss: 1.0493, val_acc: 0.6353
Epoch[33], val_loss: 1.0363, val_acc: 0.6396
Epoch[34], val_loss: 1.0406, val_acc: 0.6460
→ New best val_acc during v5 fine-tuning = 0.6460  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[35], val_loss: 1.0449, val_acc: 0.6361
Epoch[36], val_loss: 1.0438, val_acc: 0.6334
Epoch[37], val_loss: 1.0370, val_acc: 0.6468
→ New best val_acc during v5 fine-tuning = 0.6468  (ckpt saved as model_efficientnetb0_v5_tuned.pth)
Epoch[38], val_loss: 1.0554, val_acc: 0.6387
Epoch[39], val_loss: 1.0445, val_acc: 0.6369
Epoch[40], val_loss: 1.0412, val_acc: 0.6383
Epoch[41], val_loss: 1.0480, val_acc: 0.6377
Epoch[42], val_loss: 1.0471, val_acc: 0.6356
Epoch[43], val_loss: 1.0508, val_acc: 0.6334
Epoch[44], val_loss: 1.0416, val_acc: 0.6452
Epoch[45], val_loss: 1.0489, val_acc: 0.6425
Epoch[46], val_loss: 1.0528, val_acc: 0.6430
Epoch[47], val_loss: 1.0471, val_acc: 0.6438
Epoch[48], val_loss: 1.0541, val_acc: 0.6366
Epoch[49], val_loss: 1.0545, val_acc: 0.6430
Epoch[50], val_loss: 1.0464, val_acc: 0.6438
Epoch[51], val_loss: 1.0482, val_acc: 0.6465
Epoch[52], val_loss: 1.0542, val_acc: 0.6430
Epoch[53], val_loss: 1.0564, val_acc: 0.6356
Epoch[54], val_loss: 1.0548, val_acc: 0.6345
Epoch[55], val_loss: 1.0541, val_acc: 0.6411
Epoch[56], val_loss: 1.0524, val_acc: 0.6465
Epoch[57], val_loss: 1.0489, val_acc: 0.6435
Early stopping during v5 fine-tuning at epoch 58 (no val_acc gain for 20 epochs).
Fine-tuning for v5 finished.
```

Test

In [ ]:
```
from fvcore.nn import FlopCountAnalysis
# ———————— final test ——————————————————————————————
# model_eff_b0_v5.load_state_dict(torch.load('model_efficientnetb0_v5.pth')) # 0.5830681324005127
model_eff_b0_v5.load_state_dict(torch.load('model_efficientnetb0_v5_tuned.pth')) # 0.6468508839607239

# 6) Final evaluation on the test set
result_eff_test = evaluate(model_eff_b0_v5, test_dl)
print("EfficientNet-B0 Train Transform Test result:", result_eff_test)

# 7) Compute FLOPs and efficiency
dummy_input = torch.randn(1, 1, 48, 48).to(device)
flops_eff = FlopCountAnalysis(model_eff_b0_v5.eval(), dummy_input)
gflops_eff = flops_eff.total() / 1e9
acc_eff = result_eff_test['val_acc']
efficiency_eff = acc_eff / gflops_eff
print(f"EfficientNet-B0 Train Transform GFLOPs: {gflops_eff}")
print(f"EfficientNet-B0 Train Transform Accuracy: {acc_eff}")
print(f"EfficientNet-B0 Train Transform Efficiency: {efficiency_eff}")
```

```
EfficientNet-B0 Train Transform Test result: {'val_loss': 1.0023021697998047, 'val_acc': 0.6468508839607239}
Unsupported operator aten::silu_ encountered 49 time(s)
Unsupported operator aten::sigmoid encountered 16 time(s)
Unsupported operator aten::mul encountered 16 time(s)
Unsupported operator aten::add_ encountered 9 time(s)
Unsupported operator aten::dropout_ encountered 1 time(s)
The following submodules of the model were never called during the trace of the graph. They may be unused, or th
ey were accessed by direct calls to .forward() or via other python methods. In the latter case they will have ze
ros for statistics, though their statistics will still contribute to their parent calling module.
model.features.1.0.stochastic_depth, model.features.2.0.stochastic_depth, model.features.2.1.stochastic_depth, m
odel.features.3.0.stochastic_depth, model.features.3.1.stochastic_depth, model.features.4.0.stochastic_depth, mo
del.features.4.1.stochastic_depth, model.features.4.2.stochastic_depth, model.features.5.0.stochastic_depth, mod
el.features.5.1.stochastic_depth, model.features.5.2.stochastic_depth, model.features.6.0.stochastic_depth, mode
l.features.6.1.stochastic_depth, model.features.6.2.stochastic_depth, model.features.6.3.stochastic_depth, mode
l.features.7.0.stochastic_depth
EfficientNet-B0 Train Transform GFLOPs: 0.023210688
EfficientNet-B0 Train Transform Accuracy: 0.6468508839607239
EfficientNet-B0 Train Transform Efficiency: 27.868664813413712
```

Reciporical Ranking fusion

In [ ]:
```
#============================
# For V2 tuned 2 + V3 tuned + V4 tuned (3 models)
# Best weight: [0.4, 0.4, 0.2]
# Best accuracy: 0.6796

# For V2 tuned 2 + V3 tuned + V1 tuned 4 (3 models)
# Best weight: equal
# Best accuracy: 0.6887712454722764 (TARGET)

# For V2 tuned 2 + V3 tuned (2 models)
# Best weight: [0.6, 0.4]
# Best accuracy: 0.6712

# V2 tuned 2 + V3 tuned + V4 tuned + V1 tuned 4 (4 models)
# Best weight: [0.4, 0.2, 0.2, 0.2]
# Best accuracy: 0.6890

# V2 tuned + V3 tuned + V4 tuned + V1 tuned 4 (4 models)
# Best weight: [0.4, 0.2, 0.2, 0.2]
# Best accuracy: 0.6899 (TARGET)

# For V2 tuned 2 and V1 tuned 4 (2 models)
# Best weight: [0.8, 0.2]
# Best accuracy: 0.6746

# For V1 tuned 4 and V4 tuned (2 models)
# Best weight: [0.2, 0.8]
# Best accuracy: 0.6595

#============================

# Model paths - just change these paths as needed
model_paths = [
    'model_efficientnetb0_v2_tuned_2.pth',  # 0.666577160358429 (crop rotate flip - CRP)
    # 'model_efficientnetb0_tuned_4.pth',     # 0.6225019693374634 (default)
    # 'model_efficientnetb0_v3_tuned.pth',     # 0.6451432704925537 (CRP + brightness and constrast)
    # 'model_efficientnetb0_v4_tuned.pth',  # 0.6521574258804321 (CRP + cut out)
    'model_efficientnetb0_v5_tuned.pth'  # 0.6521574258804321 (CRP + cut out)
]
```

```
# Load models
models = []
for path in model_paths:
    model = to_device(EfficientNetB0(7,1), device)
    model.load_state_dict(torch.load(path))
    models.append(model)


print("RRF acc: \n")
print(evaluate_rrf(models, test_dl))

# Grid search for RRF weights
print("\n" + "="*50)
print("Grid Search for RRF Weights")
print("="*50)

# Generate weight combinations based on number of models
weights = generate_weight_combinations(len(models), step=0.05, min_weight=0.3)

best_acc = 0
best_weight = []

print(f"{'Weight':<20} {'Accuracy':<10}")
print("-" * 30)

for weight in weights:
    # Evaluate with current weight
    result = evaluate_rrf(models, test_dl, weights=weight)
    acc = result['val_acc']

    print(f"{str(weight):<20} {acc:<10.4f}")

    if acc > best_acc:
        best_acc = acc
        best_weight = weight

print("-" * 30)
print(f"Best weight: {best_weight}")
print(f"Best accuracy: {best_acc:.4f}")
```

```
RRF acc:

{'val_loss': 0.98398756980896, 'val_acc': 0.6706603510727223}

==================================================
Grid Search for RRF Weights
==================================================
Weight               Accuracy
------------------------------
[0.3, 0.7]           0.6604
[0.35, 0.65]         0.6654
[0.4, 0.6]           0.6654
[0.45, 0.55]         0.6662
[0.5, 0.5]           0.6707
[0.55, 0.45]         0.6773
[0.6, 0.4]           0.6771
[0.65, 0.35]         0.6773
[0.7, 0.3]           0.6701
------------------------------
Best weight: [0.55, 0.45]
Best accuracy: 0.6773
```

Weighted Average of Probabilities

In [ ]:

```
#============================
# 2 MODELS TIER
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_v2_tuned_2.pth
#   Model 1: model_efficientnetb0_v5_tuned.pth

# Best weight: [0.45, 0.55]
# Best accuracy: 0.6874 (TARGET)
#-------------
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
#   Model 1: model_efficientnetb0_v2_tuned_2.pth

# Best weight: [0.3, 0.7]
# Best accuracy: 0.6818 (2nd best)
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
```

```
#   Model 1: model_efficientnetb0_v3_tuned.pth

# Best weight: [0.3, 0.7]
# Best accuracy: 0.6643
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
#   Model 1: model_efficientnetb0_v4_tuned.pth

# Best weight: [0.4, 0.6]
# Best accuracy: 0.6760
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_v2_tuned_2.pth
#   Model 1: model_efficientnetb0_v3_tuned.pth

# Best weight: [0.5, 0.5]
# Best accuracy: 0.6807
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_v2_tuned_2.pth
#   Model 1: model_efficientnetb0_v4_tuned.pth

# Best weight: [0.4, 0.6]
# Best accuracy: 0.6751
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_v3_tuned.pth
#   Model 1: model_efficientnetb0_v4_tuned.pth

# Best weight: [0.4, 0.6]
# Best accuracy: 0.6746
#==================================================
# 3 MODELS TIER
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_3.pth
#   Model 1: model_efficientnetb0_v2_tuned_2.pth
#   Model 2: model_efficientnetb0_v5_tuned.pth

# Best weight: [0.2, 0.4, 0.4]
# Best accuracy: 0.6946
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
#   Model 1: model_efficientnetb0_v2_tuned_2.pth
#   Model 2: model_efficientnetb0_v5_tuned.pth

# Best weight: [0.3, 0.4, 0.3]
# Best accuracy: 0.6943 (TOP 1)
#--------------------
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
#   Model 1: model_efficientnetb0_v2_tuned_2.pth
#   Model 2: model_efficientnetb0_v5_tuned_2.pth

# Best weight: [0.2, 0.4, 0.4]
# Best accuracy: 0.6935
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
#   Model 1: model_efficientnetb0_v2_tuned_2.pth
#   Model 2: model_efficientnetb0_v3_tuned.pth

# Best weight: [0.3, 0.5, 0.2]
# Best accuracy: 0.6927 (TOP 2)
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
#   Model 1: model_efficientnetb0_v2_tuned_2.pth
#   Model 2: model_efficientnetb0_v4_tuned.pth

# Best weight: [0.3, 0.5, 0.2]
# Best accuracy: 0.6904
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
#   Model 1: model_efficientnetb0_v3_tuned.pth
#   Model 2: model_efficientnetb0_v4_tuned.pth

# Best weight: [0.2, 0.4, 0.4]
# Best accuracy: 0.6879
#--------------
# Model paths in order:
#   Model 0: model_efficientnetb0_v2_tuned_2.pth
#   Model 1: model_efficientnetb0_v3_tuned.pth
#   Model 2: model_efficientnetb0_v4_tuned.pth
```

```
# Best weight: [0.3, 0.4, 0.3]
# Best accuracy: 0.6835
#==========================
# 4 MODELS
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
#   Model 1: model_efficientnetb0_v2_tuned_2.pth
#   Model 2: model_efficientnetb0_v3_tuned.pth
#   Model 3: model_efficientnetb0_v4_tuned.pth

# Best weight: [0.3, 0.4, 0.2, 0.1]
# Best accuracy: 0.6946
#----------------------
# Model paths in order:
#   Model 0: model_efficientnetb0_tuned_4.pth
#   Model 1: model_efficientnetb0_v2_tuned_2.pth
#   Model 2: model_efficientnetb0_v3_tuned.pth
#   Model 3: model_efficientnetb0_v4_tuned.pth
#   Model 4: model_efficientnetb0_v5_tuned.pth

# Best weight: [0.1, 0.3, 0.1, 0.2, 0.3]
# Best accuracy: 0.6988
#==================================================
# Model paths - just change these paths as needed
model_paths = [
    # 'model_efficientnetb0_tuned_4.pth',     # 0.6225019693374634 (default)
    'model_efficientnetb0_v2_tuned_2.pth',  # 0.666577160358429 (crop + rotate + flip/CRP)
    # 'model_efficientnetb0_v3_tuned.pth',     # 0.6451432704925537 (CRP + brightness and constrast)
    # 'model_efficientnetb0_v4_tuned.pth',  # 0.6521574258804321 (CRP + c1t out)
    'model_efficientnetb0_v5_tuned.pth' # 0.6468508839607239 (MixUp CutMix)
]


# Load models
models = []
for path in model_paths:
    model = to_device(EfficientNetB0(7,1), device)
    model.load_state_dict(torch.load(path))
    models.append(model)


print("WAP acc: \n")
print(evaluate_weighted_average_probs(models, test_dl))

# # Grid search for WAP weights
print("\n" + "="*50)
print("Grid Search for WAP Weights")
print("="*50)



# Generate weight combinations based on number of models
weights = generate_weight_combinations(len(models), step=0.05, min_weight=0.4)

best_acc = 0
best_weight = []

print(f"{'Weight':<20} {'Accuracy':<10}")
print("-" * 30)

for weight in weights:
    # Evaluate with current weight
    result = evaluate_weighted_average_probs(models, test_dl, weights=weight)
    acc = result['val_acc']

    print(f"{str(weight):<20} {acc:<10.4f}")

    if acc > best_acc:
        best_acc = acc
        best_weight = weight

print("-" * 30)
print(f"Model paths in order:")
for i, model_path in enumerate(model_paths):
    print(f"  Model {i}: {model_path}")
print()

print(f"Best weight: {best_weight}")
print(f"Best accuracy: {best_acc:.4f}")
```

```
WAP acc:

{'val_loss': 0.98398756980896, 'val_acc': 0.6862635831707997}

================================================
Grid Search for WAP Weights
================================================
Weight               Accuracy
------------------------------
[0.4, 0.6]           0.6812
[0.45, 0.55]         0.6874
[0.5, 0.5]           0.6863
[0.55, 0.45]         0.6818
[0.6, 0.4]           0.6846
------------------------------
Model paths in order:
  Model 0: model_efficientnetb0_v2_tuned_2.pth
  Model 1: model_efficientnetb0_v5_tuned.pth

Best weight: [0.45, 0.55]
Best accuracy: 0.6874
```

Since the final model is an ensemble of two EfficientNetb0, the gflops will be approximately 2 times EfficientNetb0 gflops, which is 0.046421376

In [ ]:
```python
# final accuracy test for submitted model
model_paths = [
    'model_efficientnetb0_v2_tuned_2.pth',  # 0.666577160358429 (crop + rotate + flip/CRP)
    'model_efficientnetb0_v5_tuned.pth' # 0.6468508839607239 (MixUp CutMix)
]


# Load models
models = []
for path in model_paths:
    model = to_device(EfficientNetB0(7,1), device)
    model.load_state_dict(torch.load(path))
    models.append(model)

evaluate_weighted_average_probs(models, test_dl, weights=[0.45,0.55])
```

```
{'val_loss': 0.98398756980896, 'val_acc': 0.6873780997492338}
```

In [ ]:
```python
# Final accuracy test for best accuracy ever achieved
model_paths = [
    'model_efficientnetb0_tuned_4.pth',     # 0.6225019693374634 (default)
    'model_efficientnetb0_v2_tuned_2.pth',  # 0.666577160358429 (crop + rotate + flip/CRP)
    'model_efficientnetb0_v3_tuned.pth',     # 0.6451432704925537 (CRP + brightness and constrast)
    'model_efficientnetb0_v4_tuned.pth',  # 0.6521574258804321 (CRP + cut out)
    'model_efficientnetb0_v5_tuned.pth' # 0.6468508839607239 (MixUp CutMix)
]


# Load models
models = []
for path in model_paths:
    model = to_device(EfficientNetB0(7,1), device)
    model.load_state_dict(torch.load(path))
    models.append(model)


print("WAP acc: \n")
print(evaluate_weighted_average_probs(models, test_dl, weights=[0.1, 0.3, 0.1, 0.2, 0.3]))
```

```
WAP acc:

{'val_loss': 1.0393049359321593, 'val_acc': 0.6988018946781833}
```