Classification of facial expressions using custom CNNs and pre-trained models

Roberta Angioni, 60/73/65312
Mattia Cani, 60/73/65306

The task

The task addressed in this project involves the classification of images from the FER2013 dataset (https://www.kaggle.com/datasets/msambare/fer2013) based on the facial expressions they represent. Specifically, a custom convolutional neural network trained from scratch and the ResNet18 model are used. So, transfer learning is leveraged through the ResNet18 model.

However, it is important to note that although two types of models are used, three models are utilized and tested in the project. This is because ResNet is trained once for 5 epochs, and then the tests are repeated by training it for 10 epochs. This approach ensures a fair comparison with the CNN, as it is trained for 10 epochs.

This classification exercise is approached in two different versions: binary, considering only the classes related to happy and sad emotions, and multiclass, where the classes related to anger and fear are also included. Both versions are subjected to the same tests conducted using all three models, which are evaluated according to the following metrics: accuracy, recall, precision and F1 score, considered both on individual classes and in general. All tests are done twice, in order to compare the performances obtained with two different loss functions: cross-entropy loss and focal loss.

The programming language used is Python.

Theoretical background

Convolutional Neural Network

CNNs are neural network architectures designed to process data with a grid-like topology, such as images. They utilize convolutional layers to automatically and adaptively learn spatial hierarchies of features from input data. Key components of CNNs include:

- **Convolutional Layers**: Extract features by applying filters (kernels) to input data.
- **Pooling Layers**: Downsample feature maps to reduce dimensionality and computational cost while preserving key information.
- **Fully Connected Layers**: Combine learned features for classification tasks.
- **Activation Functions**: Introduce non-linearity to enhance learning capacity.

In the context of this project, the CNN used is defined using `PyTorch`'s `nn.Module` class and follows the typical structure:

- **Three Convolutional Layers:**
    - The first one takes the input image and applies 32 filters of size 3x3. Padding is applied to maintain the spatial dimensions of the output. A batch

normalization layer follows the convolution to stabilize training and speed up convergence.
    - The second one receives the output from the first layer and applies 64 filters of size 3x3. It also uses batch normalization for further refinement.
    - The last one receives the output from the second layer and applies 128 filters of size 3x3. As in the previous layers, batch normalization is applied.

- **Pooling Layer:** after the convolutional block, a max-pooling operation with a kernel size of 2x2 is applied. This reduces the spatial dimensions of the feature map, effectively downsampling the data and reducing the number of parameters and computational cost.

- **Dropout Layer:** to mitigate overfitting, a dropout layer is included after the convolutional ones. Dropout randomly sets a fraction (50% in this case) of the input units to zero during training, which helps prevent the model from becoming too reliant on certain features.

- **Two Fully Connected Layers:**

    - The first one has 512 units, and it takes the flattened output from the convolutional layers. A ReLU activation function is applied to introduce non-linearity and help learn complex patterns.
    - The second fully connected layer outputs the final predictions, so it produces a vector of raw scores for each class. The number of units in this layer corresponds to the number of classes.

ResNet

ResNet18 is part of the ResNet family (Residual Networks) and is a deep pre-trained convolutional neural network that efficiently addresses vanishing gradient issues in deep learning.

- **Residual Blocks**: ResNet uses skip connections to allow gradients to flow directly through the network, enabling deeper architectures without performance degradation.
- **Pre-trained Weights**: the model is initially trained on a large dataset and fine-tuned on the dataset used for emotion recognition.

ResNet18 has 18 layers in total. In particular, the structure is roughly composed as it follows:

1. **Initial Convolution and Pooling:** a first convolutional layer with max pooling.
2. **Residual Blocks:** four blocks with two 3x3 convolutions each and increasing filters.
3. **Global Average Pooling**

4. **Fully Connected Layer**: computes raw scores for classification tasks (by default, it works with 1000 output classes)
5. **Softmax activation**: transforms raw scores into probabilities

EfficientNet

EfficientNet-B1 is part of the EfficientNet family, a series of convolutional neural networks designed to achieve high accuracy while maintaining computational efficiency through a balanced scaling approach.

- **Optimized scaling**: it uses compound scaling, a method that proportionally balances depth (number of layers), width (number of channels per layer), and input resolution. Unlike traditional models that scale only one dimension (like making the network deeper or wider), compound scaling improves performance without excessive computational cost by optimizing all three aspects simultaneously.
- **MBConv Blocks**: it uses Mobile Inverted Bottleneck Convolution blocks, derived from depthwise separable convolutions. They improve efficiency by using pointwise convolutions to reduce dimensionality, followed by depthwise (separate per-channel) convolutions for feature extraction.
- Includes **skip connections** similar to ResNet.
- **Pre-trained Weights**: in the same way as ResNet, the model is pre-trained on a large general-purpose dataset and can be fine-tuned for specific tasks.

The structure is composed as it follows:

1. **Initial Layer**: convolutional layer.
2. **MBConv Blocks**: 16 blocks with increasing filters (up to 320 filters are applied in the last block). Kernel size could be 3x3 or 5x5 based on the block and it's the same with the stride, that can be 1 or 2.
3. Final **pointwise convolutional layer** with 1280 filters.
4. **Global Average Pooling**
5. **Fully Connected Layer**: as for ResNet, it computes raw scores for classification tasks with the same number of output classes by default.
6. **Softmax activation** to obtain probabilities.

MobileNet

MobileNetV2 is part of the MobileNet family, which is composed of convolutional neural networks designed for deep learning on mobile and edge devices. It achieves a balance between high accuracy and low computational cost by using optimized architectural components.

- **Optimized Efficiency:** it is designed with a lightweight structure, reducing the number of parameters while maintaining performance. It introduces linear bottleneck

layers and inverted residual connections, which improve information flow and reduce memory usage.

- **MBConv Blocks:** the architecture is based on Mobile Inverted Bottleneck Convolution blocks, similar to EfficientNet.
- **Pre-trained Weights:** like ResNet and EfficientNet, MobileNetV2 can be pre-trained on large general-purpose datasets and later fine-tuned for specific tasks.

The model has the following structure:

1. **Initial Layer**: convolutional layer.
2. **MBConv Blocks**: 17 blocks, progressively increasing the number of filters (up to 320 filters in the last stage). Expansion factors are applied to increase feature representation before depthwise convolutions. Kernel size is 3x3 and the stride can be 1 or 2 depending on the stage.
3. **Pointwise convolution** with 1280 filters.
4. **Global average pooling**
5. **Fully Connected Layer:** computes raw scores for classification tasks.
6. **Softmax Activation** to produce class probabilities.

Cross-Entropy Loss

It is a widely used loss function for classification tasks, particularly when dealing with multi-class problems. It measures the difference between the predicted probability distribution (output of the model) and the true probability distribution (ground truth labels).
In particular, this loss is calculated as the negative logarithm of the predicted probability assigned to the correct class. This encourages the model to assign higher probabilities to the correct class during training.

$$H(P^* \mid P) = -\sum_i \underbrace{P^*(i)}_{\text{TRUE CLASS DISTIRBUTION}} \log \underbrace{P(i)}_{\text{PREDICTED CLASS DISTIRBUTION}}$$

Focal Loss

It is a loss function designed to address the class imbalance problem in classification tasks, especially in scenarios where some classes are underrepresented. Actually, the focal loss is a modification of the standard cross-entropy loss that down-weights the loss for well-classified examples and focuses more on hard-to-classify examples.

$$\mathrm{FL}(p_\mathrm{t}) = -\alpha_\mathrm{t}(1-p_\mathrm{t})^\gamma \log(p_\mathrm{t}).$$

where p_t is the probability for the correct class, α is a class balancing factor, γ increases emphasis on the classes that the model finds difficult.

So, focal loss ensures that the model focuses more on improving the classification of hard examples while minimizing the effect of well-classified ones.

Preprocessing

The reason why the multiclass version of the dataset used includes only 4 classes – instead of the original 7 – lies in the attempt to reduce the computational load in favor of faster performance and more accessible testing. For the same reason, and to ensure a balanced dataset, the number of images per class used during the training phases was reduced to approximately 4000.

Additional preprocessing techniques used include resizing the images – 48x48 for use with the CNN and 224x224 in the case of ResNet – and their normalization.

Training

Training the CNN

The `train_model_CNN` function is designed to train the CNN while tracking and saving the training and validation performance across multiple epochs. The function handles the full training process, including the forward pass, loss calculation, backpropagation, and parameter updates. Additionally, it computes accuracy metrics for both the training and validation sets and saves the results in a CSV file for later analysis.

The training process is carried out by a trained loop, so for each epoch the model is set in training mode, then the following main steps are performed for each batch:

1. The optimizer gradients are reset.

2. The model makes predictions by passing the images through the forward pass, producing the raw output scores.
3. The loss is computed by comparing the model's output with the ground truth labels using the specified loss function.
4. Backpropagation is performed and calculates the gradients.
5. The optimizer updates the model's parameters based on the gradients calculated in the previous step.
6. The running loss is accumulated to track how well the model is performing during the training phase.

After completing the training, the `validate_model` function is called to evaluate the model's performance on the validation dataset, so this function computes the validation accuracy.

```
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()  # Reset gradients
        outputs = model(images)  # Pass through the model
        loss = criterion(outputs, labels)  # Calculate loss
        loss.backward()  # Calculate gradients
        optimizer.step()  # Update weights

        running_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    train_loss = running_loss / len(train_loader)
    train_accuracy = 100 * correct / total

    # Validate
    val_accuracy = validate_model(model, val_loader, device)

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {train_loss:.4f}, "
          f"Accuracy: {train_accuracy:.2f}%, Validation Accuracy: {val_accuracy:.2f}%")
```

training phase

Fine-tuning the pre-trained models

In the same way, the `train_model_TL` function is responsible for training the pre-trained models. It is trivial that the training process in this case is not made from scratch, only the last layers are fine-tuned for our specific task.

The function starts by freezing the parameters of all layers in the model. This is done to retain the pre-trained feature extraction capabilities of the earlier layers while preventing them from being updated during training, then the last two layers are released to adjust them to learn the task-specific features. Correggere con dataset finale perchè il numero di layer da sbloccare potrebbe cambiare

The training loop performs exactly the same steps that are performed during the CNN training, and at the end of the computation the results are saved in a CSV file.

```python
def freeze_resnet_layers(model):

    # Freeze all layers
    for param in model.parameters():
        param.requires_grad = False

    # Unfreeze the penultimate
    for param in model.layer4.parameters():
        param.requires_grad = True

    # Unfreeze the last
    for param in model.fc.parameters():
        param.requires_grad = True
```

This function is called at the start to freeze and adjust layers.

Testing

Testing the CNN

For convenience reasons, the testing part is carried out by the `train_and_test_CNN` that, as the name suggests, handles the entire workflow for training and validating – by calling the functions previously explained – and for testing our CNN model. Here's a high-level breakdown of its functionality:

- A series of transformations for the input images, including resizing, tensor conversion and normalization are performed to all the dataset.
- The function splits the training dataset into a training and validation set (80% for training, 20% for validation).
- Based on the classification mode (binary or multiclass), the function initializes a CNN model with the appropriate number of output classes.
- Based on the parameters with which it is called, the function chooses the loss function and the optimizer to use.
- The model is then trained and saved in a .pth file.
- The saved model is loaded and passed to the `evaluate_model` function that will perform the actual test on the test dataset and save the results in a CSV file.

The `evaluate_model` function, for each batch of test images, performs a forward pass to obtain predictions, updates the total and class-specific correct predictions and populates the confusion matrix.

```python
with torch.no_grad():
    for images, labels in test_loader:

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

        # Perform a forward pass
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        # Update total and correct predictions
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        # confusion matrix update
        for i in range(len(labels)):
            true_label = labels[i].item()
            pred_label = predicted[i].item()
            class_total[true_label] += 1
            if true_label == pred_label:
                class_correct[true_label] += 1
            confusion_matrix[true_label][pred_label] += 1
```

evaluate_model's main loop. The confusion matrix is then used to calculate metrics

Testing the pre-trained models

The `train_and_test_TL` function is designed to train and evaluate our pre-trained models and, as for the correspondent function for the custom CNN, it supports both binary and multiclass classification and integrates the entire pipeline, including preprocessing, training, validation, and testing that are carried out in a very similar way as in the correspondent function for the CNN. The function saves the trained model in a .pth file and the results in a CSV file.

Before the training of the model, based on which parameter is passed to the function, the corresponding pre-trained model is loaded with its default weights, the final fully connected layer is adjusted to match the number of classes based on the mode.

```python
if model_name == "resnet":
    model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
    num_features = model.fc.in_features
elif model_name == "efficientnet":
    model = models.efficientnet_b1(weights=models.EfficientNet_B1_Weights.DEFAULT)
    num_features = model.classifier[1].in_features
elif model_name == "mobilenet":
    model = models.mobilenet_v2(weights=models.MobileNet_V2_Weights.DEFAULT)
    num_features = model.classifier[1].in_features


if mode == "binary":
    if model_name == "resnet":
        model.fc = nn.Linear(num_features, 2)
    elif model_name == "efficientnet":
        model.classifier[1] = nn.Linear(num_features, 2)
    elif model_name == "mobilenet":
        model.classifier[1] = nn.Linear(num_features, 2)
    weights = torch.tensor([1.0, 1.5], dtype=torch.float32)
elif mode == "multiclass":
    if model_name == "resnet":
        model.fc = nn.Linear(num_features, 4)
    elif model_name == "efficientnet":
        model.classifier[1] = nn.Linear(num_features, 4)
    elif model_name == "mobilenet":
        model.classifier[1] = nn.Linear(num_features, 4)
    weights = torch.tensor([2.0, 1.5, 1.0, 1.3], dtype=torch.float32)
```

Load and adjust model.

Evaluation

This paragraph will present and discuss the results obtained during the testing phase. The following bullet list shows how the classes have been labeled:

- Binary task
  - 0 = happy
  - 1 = sad
- Multiclass task
  - 0 = angry
  - 1 = fear
  - 2 = happy
  - 3 = sad

CNN

test_binario_crossentropy

Metric	Value
Test Accuracy	86.36213174445548
Macro Precision	85.8494860144367
Macro Recall	86.20804926185868
Macro F1 Score	86.00741083835109
Accuracy Class 0	87.0913190529876
Accuracy Class 1	85.32477947072975
Precision Class 0	89.40972222222221
Precision Class 1	82.2892498066512
Recall Class 0	87.0913190529876
Recall Class 1	85.32477947072975
F1 Score Class 0	88.23529411764707
F1 Score Class 1	83.77952755905513

test_binario_focal

Metric	Value
Test Accuracy	86.09731876861966
Macro Precision	85.56995101287076
Macro Recall	86.25653089398773
Macro F1 Score	85.81390751815272
Accuracy Class 0	85.34385569334836
Accuracy Class 1	87.1692060946271
Precision Class 0	90.44205495818399
Precision Class 1	80.69784706755753
Recall Class 0	85.34385569334837
Recall Class 1	87.1692060946271
F1 Score Class 0	87.81902552204177
F1 Score Class 1	83.80878951426368

test_multiclasse_crossentropy

Metric	Value
Test Accuracy	60.743553867679395
Macro Precision	64.04498053142007
Macro Recall	56.73162993640146
Macro F1 Score	56.697578493716144
Top-2 Accuracy	80.65160903457925
Accuracy Class 0	36.74321503131524
Accuracy Class 1	33.203125
Accuracy Class 2	73.90078917700113
Accuracy Class 3	83.0793905372895
Precision Class 0	63.883847549909255
Precision Class 1	60.82289803220036
Precision Class 2	88.52126941255908
Precision Class 3	42.95190713101161
Recall Class 0	36.74321503131524
Recall Class 1	33.203125
Recall Class 2	73.90078917700113
Recall Class 3	83.0793905372895
F1 Score Class 0	46.653412856196155
F1 Score Class 1	42.95641187618446
F1 Score Class 2	80.55299539170507
F1 Score Class 3	56.62749385077891

test_multiclasse_focal

Metric	Value
Test Accuracy	63.42194683190086
Macro Precision	61.17497132054993
Macro Recall	60.29575336057437
Macro F1 Score	60.35966181729796
Top-2 Accuracy	82.53048171097342
Accuracy Class 0	55.11482254697286
Accuracy Class 1	42.48046875
Accuracy Class 2	79.59413754227734
Accuracy Class 3	63.99358460304731
Precision Class 0	55.87301587301587
Precision Class 1	52.79126213592234
Precision Class 2	85.78371810449575
Precision Class 3	50.25188916876574
Recall Class 0	55.11482254697286
Recall Class 1	42.48046875
Recall Class 2	79.59413754227734
Recall Class 3	63.99358460304732
F1 Score Class 0	55.491329479768794
F1 Score Class 1	47.07792207792208
F1 Score Class 2	82.57309941520468
F1 Score Class 3	56.2962962962963

ResNet - training for 5 epochs

test_binario_crossentropy

Metric	Value
Test Accuracy	88.01721284342933
Macro Precision	87.68201324307796
Macro Recall	88.7726936982467
Macro F1 Score	87.86622066750755
Accuracy Class 0	84.44193912063135
Accuracy Class 1	93.10344827586206
Precision Class 0	94.57070707070707
Precision Class 1	80.79331941544885
Recall Class 0	84.44193912063133
Recall Class 1	93.10344827586206
F1 Score Class 0	89.21977367480643
F1 Score Class 1	86.51266766020865

test_binario_focal

Metric	Value
Test Accuracy	88.28202581926514
Macro Precision	87.77997880724467
Macro Recall	88.39069460052491
Macro F1 Score	88.01973689516942
Accuracy Class 0	87.76775648252537
Accuracy Class 1	89.01363271852446
Precision Class 0	91.91263282172373
Precision Class 1	83.64732479276563
Recall Class 0	87.76775648252536
Recall Class 1	89.01363271852446
F1 Score Class 0	89.79238754325259
F1 Score Class 1	86.24708624708624

test_multiclasse_crossentropy

Metric	Value
Test Accuracy	57.30561663002199
Macro Precision	62.78204837691487
Macro Recall	56.858477207865384
Macro F1 Score	54.98893973854535
Top-2 Accuracy	79.39236458125124
Accuracy Class 0	81.73277661795407
Accuracy Class 1	28.22265625
Accuracy Class 2	62.62683201803833
Accuracy Class 3	54.85164394546913
Precision Class 0	38.1207400194742
Precision Class 1	66.74364896073904
Precision Class 2	95.85849870578085
Precision Class 3	50.405305821665436
Recall Class 0	81.73277661795407
Recall Class 1	28.22265625
Recall Class 2	62.626832018038336
Recall Class 3	54.85164394546913
F1 Score Class 0	51.99203187250997
F1 Score Class 1	39.67055593685656
F1 Score Class 2	75.75860893283328
F1 Score Class 3	52.53456221198157

test_multiclasse_focal

Metric	Value
Test Accuracy	63.341994803118126
Macro Precision	62.7198865509418
Macro Recall	58.3220082105106
Macro F1 Score	57.935868146735565
Top-2 Accuracy	83.60983409954028
Accuracy Class 0	45.82463465553236
Accuracy Class 1	28.41796875
Accuracy Class 2	86.47125140924464
Accuracy Class 3	72.57417802726543
Precision Class 0	60.136986301369866
Precision Class 1	64.23841059602648
Precision Class 2	76.96939287506271
Precision Class 3	49.53475643130816
Recall Class 0	45.82463465553236
Recall Class 1	28.41796875
Recall Class 2	86.47125140924464
Recall Class 3	72.57417802726543
F1 Score Class 0	52.014218009478675
F1 Score Class 1	39.404197698036555
F1 Score Class 2	81.44411998938146
F1 Score Class 3	58.88093689004555

ResNet - training for 10 epochs

test_binario_crossentropy

Metric	Value
Test Accuracy	88.18272095332671
Macro Precision	87.69131490504162
Macro Recall	88.5086326687997
Macro F1 Score	87.96097402409664
Accuracy Class 0	86.64036076662909
Accuracy Class 1	90.37690457097032
Precision Class 0	92.75799637899819
Precision Class 1	82.62463343108504
Recall Class 0	86.64036076662909
Recall Class 1	90.37690457097032
F1 Score Class 0	89.59487030020401
F1 Score Class 1	86.32707774798928

test_binario_focal

Metric	Value
Test Accuracy	84.50844091360477
Macro Precision	84.88350250041916
Macro Recall	85.92803110780417
Macro F1 Score	84.4353643360468
Accuracy Class 0	77.7903043968433
Accuracy Class 1	94.06575781876504
Precision Class 0	94.91059147180194
Precision Class 1	74.85641352903637
Recall Class 0	77.7903043968433
Recall Class 1	94.06575781876504
F1 Score Class 0	85.50185873605949
F1 Score Class 1	83.3688699360341

test_multiclasse_crossentropy

Metric	Value
Test Accuracy	59.304417349590246
Macro Precision	61.62194260656547
Macro Recall	56.93016056115607
Macro F1 Score	54.78496168940568
Top-2 Accuracy	80.19188486907855
Accuracy Class 0	81.00208768267224
Accuracy Class 1	32.421875
Accuracy Class 2	82.2998872604284
Accuracy Class 3	31.996792301523655
Precision Class 0	36.74242424242424
Precision Class 1	62.99810246679317
Precision Class 2	84.98253783469151
Precision Class 3	61.76470588235294
Recall Class 0	81.00208768267223
Recall Class 1	32.421875
Recall Class 2	82.2998872604284
Recall Class 3	31.99679230152366
F1 Score Class 0	50.553745928338756
F1 Score Class 1	42.81108961960026
F1 Score Class 2	83.6197021764032
F1 Score Class 3	42.15530903328051

test_multiclasse_focal

Metric	Value
Test Accuracy	60.18388966620028
Macro Precision	59.26289745282824
Macro Recall	58.43937236776649
Macro F1 Score	57.60737034497505
Top-2 Accuracy	79.97201678992604
Accuracy Class 0	67.5365344467641
Accuracy Class 1	48.046875
Accuracy Class 2	75.59188275084554
Accuracy Class 3	42.582197273456295
Precision Class 0	40.949367088607595
Precision Class 1	49.596774193548384
Precision Class 2	88.28176431863068
Precision Class 3	58.223684210526315
Recall Class 0	67.5365344467641
Recall Class 1	48.046875
Recall Class 2	75.59188275084556
Recall Class 3	42.582197273456295
F1 Score Class 0	50.985027580772254
F1 Score Class 1	48.80952380952381
F1 Score Class 2	81.44549043425448
F1 Score Class 3	49.18943955534969

Comment on the results

All the tests show good results in the **binary task**, with an acceptable balance between classes in most cases. Using **focal loss** in the computation led to little to no general improvement in this task, while for the multiclass task it brought some improvement in all tests, with respect to the test accuracy at least.

Using focal loss on a balanced dataset for a binary classification task may seem counterintuitive, as focal loss is primarily designed to handle imbalanced datasets or class imbalance problems. However, it can still be useful because it reduces the importance of well-classified samples (i.e., those with high predicted probabilities), focusing more on hard or uncertain samples.

In a balanced dataset, models tend to quickly converge on easier examples, ignoring the more challenging ones. This can lead to metrics like precision and recall becoming unbalanced, which can be addressed by focal loss. It forces the model to focus more on difficult samples, increasing the chances of improving the balance between precision and recall.

However, in the case of the ResNet model trained for 10 epochs, a performance decline is observed when using focal loss in the binary case. This likely occurs because, even with a standard loss function like cross-entropy, the metrics were already well balanced. This effect depends on how focal loss adjusts the weights of harder samples: it amplifies the weight of examples classified with greater uncertainty, which can lead to overfitting on samples that are considered difficult to classify for the model. So, it may excessively penalize correctly classified samples that were already well-predicted and negatively affect the trade-off between false positives and false negatives for all classes in general.

In fact, the cases when the focal loss lead to a significant improvement are the ones where it is applied to perform the multiclass classification task using the custom CNN and the ResNet model trained for 5 epochs: the same tests with the cross-entropy loss showed a bigger gap between accuracy, precision and recall among certain classes, so applying focal loss led to a more significant test accuracy improvement and better balance between the other values.

Regarding the **multiclass task**, a significant drop in overall performance is observed in comparison to the binary task. A pattern can be identified in the results: the “happy” class tends to achieve the best performance on average and seems to be the easiest one to classify.

This drop in performance may be due to various factors:

- The classes “angry,” “fear,” and “sad” have overlapping features and may be difficult to classify correctly, even though weights to make classification easier for these classes were passed to the loss functions. Even human testing of image classification revealed some difficulties in distinguishing clearly between these classes.

- The dataset may be too small to allow ResNet to train effectively, avoid overfitting and generalize properly, even though a version of the model with relatively fewer parameters (ResNet18) was chosen and techniques like weight decay were applied to the optimizer. However, this issue may be secondary to the overlapping features in the case of the multiclass task, as the performance in the binary task, where the dataset is even smaller, does not show such significant problems.
- The dataset may be too different from the domain in which ResNet was trained (generic image classification), to the point where the model might struggle to extract facial emotion features and fail using all of its capabilities. The decision to train not only the last layer but also the penultimate one was meant to mitigate this potential issue while still retaining the benefits of transfer learning.

Actually, the last two points of the bulleted list could be problems that affect both tasks: this could explain why a much more deep model than our custom CNN doesn't achieve significantly better performances throughout the whole project.

The fact that the conditions to fully leverage ResNet's potential are not met is also evident when looking at the validation results of the model trained for 10 epochs. After around the fifth epoch, the model shows clear signs of overfitting despite all the techniques applied to prevent it. In some cases, the validation accuracy is far from the training accuracy value, and while the training accuracy continues to increase, the validation accuracy stabilizes. So, the model fails to generalize properly according to its capabilities.

validation_multiclasse_focal

Epoch	Train Loss	Train Accuracy	Validation Accuracy
1	0.6612050279974937	56.752110034385744	61.331666145670525
2	0.4193992327153683	66.09878086902157	59.1434823382307
3	0.3149767840653658	71.30353235386058	63.801187871209756
4	0.20681137705221772	77.47733666770866	62.48827758674586
5	0.10688004518859089	84.42482025633011	65.14535792435136
6	0.036294346472714095	91.23944982807127	68.80275085964364
7	0.0038569496004492976	97.91341044076273	68.89653016567678
8	0.000602162730768896	99.3826195686152	69.11534854642076
9	0.00020110083466875038	99.8046264457643	69.17786808377618
10	0.00015734322795310618	99.78899656142545	69.2403876211316

Overfitting example

Below are examples of how the images from the more difficult-to-classify classes can be quite tricky. Take into account the following two images depicting men:

These two images both come from the “angry” training set, but they can be misleading. They can be mistaken for expressing fear in the first one and sadness in the second.

Another example is given by the following pictures of a toddler and a baby:

Both pictures can be interpreted as showing the same emotion, but the first one comes from the “sad” class, while the second one comes from the “fear” class.

As the images contained in the dataset are not very clear, all the tests were repeated after applying a median filter to the images to denoise them, but this did not lead to a significant change in the results of the test phase for the classification task.

However, to address this ambiguity throughout the dataset and test the models’ abilities in a proper way, also another metric has been taken into account in the case of the multiclass task: top-2 score, which measures the percentage of times the model’s correct prediction is among the two highest-probability classes assigned by the model. As can be seen from the results, the top-2 score value is always high enough to assess that models are performing in an acceptable way despite the overlapping features.