Faculty of automatic control, electronics and informatics.

Biologically Inspired Artificial Intelligence Project

Computer Vision - Facial Expression classification using CNNs. A Comparative Study of fine-tuned Convolutional Neural Networks and Transfer Learning on RESNET-18.

By:
Asser Moustafa
Michał Smaluch

Supervisor:
Dr. Kuaban Godlove

- # Intro

Facial expressions recognition plays a significant role in understanding human emotions and has numerous applications in fields such as **psychology**, **human-computer interaction (HCI)**[1] , and **surveillance systems**. Being able to automatically detect and classify facial expressions with high accuracy and precision has become a challenging yet essential task in the field of computer vision and machine learning. In this report, we present a machine learning approach utilizing Convolutional Neural Networks (CNNs) for facial expression recognition / classification.

The objective of this project is to **develop a facial expression classification system that can effectively classify facial expressions into different emotion categories,** such as happiness, sadness, anger, surprise, fear and neutral. By leveraging the power of deep learning and utilizing a dataset specifically curated for facial expression recognition, we aim to achieve high accuracy and robust performance in real-world scenarios.

In this report, we will discuss the analysis of the task, explore different approaches to solve the facial expression recognition problem, and present the chosen methodology based on CNNs. We will delve into the details of the FERPlus dataset, which serves as the foundation for training and evaluating our model. Additionally, we will provide insights into the software solution developed, including the structure of the codebase and the implementation of the graphical user interface (GUI).

By conducting experiments and analyzing the results, we aim to gain a deeper understanding of the performance of our facial expression recognition system. The findings of this project will offer valuable insights into the challenges and potential improvements in this area of research.

The following sections of this report will delve into the detailed analysis of the task, the chosen methodology, the software solution, the conducted experiments, and the overall conclusions. Let us begin by exploring the various approaches and methodologies for facial expression recognition.

● Analysis of the task

In this section, we will explore different approaches to solve the facial expression recognition problem and analyze the chosen methodology in detail.

a. Possible Approaches

Facial expression recognition can be approached using various techniques, including:

1. In **traditional machine learning**, relevant features are extracted from facial images using techniques like Histogram of Oriented Gradients, Local Binary Patterns, or Eigenfaces. A machine learning classifier, such as Support Vector Machines or Random Forests, is then trained on these extracted features.

2. **Deep learning uses Convolutional Neural Networks (CNNs)** designed specifically for image classification, which can achieve high accuracy in facial expression classification through convolutional layers. Pre-trained models like VGGNet, ResNet, or Inception, which have been trained on datasets like ImageNet, can be fine-tuned for facial expression datasets. Transfer learning can also be applied by fine-tuning pre-trained CNN models.

3. **Ensemble methods combine multiple classifiers or models,** such as Random Forests, Gradient Boosting, or Stacking, to improve classification performance by aggregating predictions from multiple models or classifiers. Facial landmarks or action units can also be detected and used for classification by extracting features related to those landmarks or units.

4. **Hybrid approaches combine multiple approaches or models**, such as traditional feature extraction methods with deep learning models or facial landmarks with deep learning models, to leverage their complementary strengths and achieve improved accuracy.

After careful consideration, **we have chosen the CNN-based deep learning approach**[8] **for our facial expression recognition task**. The flexibility and power of CNNs in learning discriminative features make them well-suited for capturing subtle variations in facial expressions, ultimately leading to improved accuracy and robustness.

b. Chosen Methodology

Our chosen methodology involves training a **CNN model on the FERPlus** dataset, and comparing the tuned model performance with a pretrained model, in our case we will work with **RESNET-18 pretrained on ImageNet**, utilizing transfer learning[17] techniques.

To train the CNN model, we employ a deep learning framework, such as **PyTorch**, which provides efficient tools for building and training neural networks. The CNN architecture consists of multiple convolutional layers, followed by pooling layers and fully connected layers. We employ appropriate activation functions, such as ReLU, and incorporate regularization techniques, such as dropout or batch normalization, to mitigate overfitting. Architecture of the network is later explained in detail.

c. Data set[3]

The FERPlus[4] dataset consists of grayscale images of faces, each labeled by 10 human taggers who each vote what they think this expression is, so the final label contains votes for each emotion category, emotion categories are : neutral, happiness, surprise, sadness, anger, disgust, fear, contempt, unknown, and NF. The dataset has been preprocessed to align and center the faces, ensuring consistent facial positioning across the images.

The Key Difference between this dataset and the well-known FER2013 set is that **the FER+ contains 10 classes**, not just 7 as in the FER2013.
It also allows for **multi-label-multi-class classification**, which we didn't pursue this time as it would be much more complicated to do multi-label classification using a simple CNN architecture.

By fine-tuning the model's hyperparameters and leveraging techniques like data augmentation, we aim to achieve high accuracy and robust generalization on unseen facial expression images.

The training set consists of 28,709 examples. The public test set (Validation set) used for the leaderboard consists of 3,589 examples. The final test set consists of another 3,589 examples.[1]

One thing to notice though is that the dataset is **not perfectly balanced** as there exists :

Emotion	Training set	Validation set	Testing set
Happiness	7526	899	928
Neutral	10294	1328	1258
Anger	2463	325	321
Sadness	3530	415	446
Fear	655	71	97
Disgust	191	32	20
Contempt	168	26	27
NF	2	1	0
Unknown	171	23	28

The limited number of samples in certain categories, such as Fear, Disgust, and Contempt, compared to other classes, impacts the performance of the network. Specifically, it enhances the network's proficiency in identifying emotions like "Happiness" while hindering its ability to recognize "Disgust." Therefore, relying solely on accuracy to evaluate the models is inadequate. Instead, metrics such as **precision, recall**[6]**, F1-Score**[5]**, and confusion matrix** are also considered to assess the models comprehensively.

This imbalance -due to an oversight on our part- was not clearly established as knowledge to us at the beginning. Which affected our path of experiments at the beginning and misleadingly led us to base our experiments solely on accuracy.
However, as it will be explained in the conclusion, it played a crucial role in our further investigation and helped us develop a deeper understanding of Neural networks.

In the next section, we will delve deeper into the Neural Network characteristics and implementation that best suits the data set.

• Internal and External specifications

In this section, we will outline the specifications of the software solution developed for the facial expression recognition project. We will discuss the Network architecture, different layers used, data augmentation techniques, normalization and regularization methods performed, as well as the user interface/GUI/console aspects of the software.

a. Network architecture

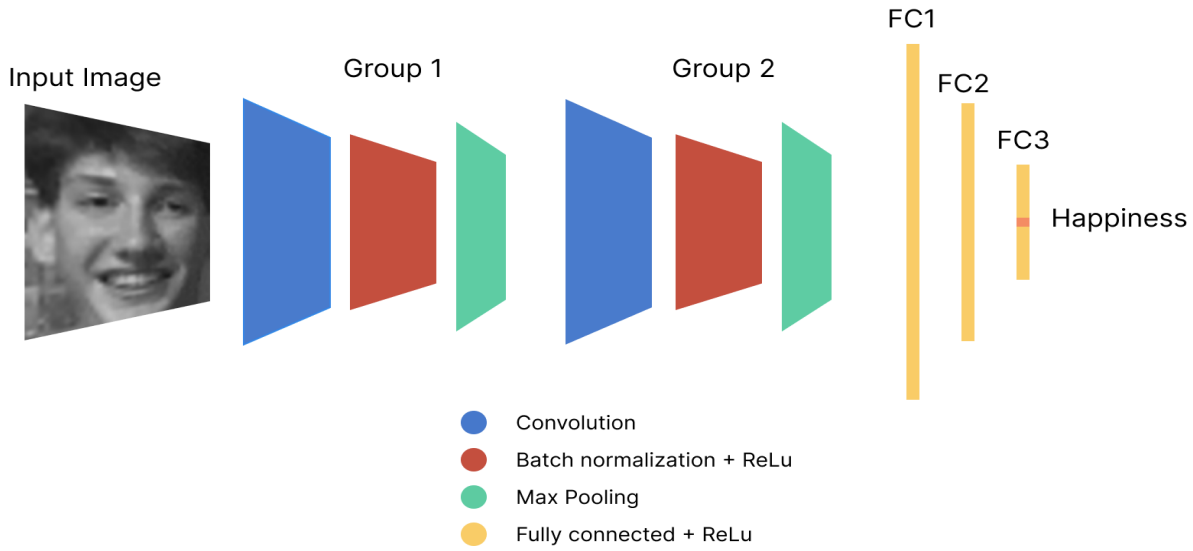


Figure 1 - Network architecture

The Network model consists of the following layers:
Input: The network expects grayscale input images of size n x n (where n is the input image size).

Convolutional Layer 1: The first convolutional layer (conv1) has 1 input channel (grayscale) and 6 output channels. It uses a kernel/filter size of 5x5, resulting in an output size of 6x44x44. This means that conv1 applies six 5x5 filters to the input image, producing six feature maps of size 44x44.

Batch Normalization 1[7]: Batch normalization (bn1) is applied after conv1 to normalize the output feature maps.

Max Pooling 1[9]: A max pooling layer is applied with a kernel size of 2x2 and a stride of 2. This reduces the spatial dimensions of the feature maps by half, resulting in output feature maps of size 6x22x22.

Convolutional Layer 2: The second convolutional layer (conv2) takes the 6 input channels from the previous layer and produces 16 output channels. It uses a kernel size of 5x5, resulting in output feature maps of size 16x18x18.

Batch Normalization 2: Batch normalization (bn2) is applied after conv2 to normalize the output feature maps.

Max Pooling 2: Another max pooling layer is applied with a kernel size of 2x2 and a stride of 2. This reduces the spatial dimensions of the feature maps by half, resulting in output feature maps of size 16x9x9.

Fully Connected Layers: The flattened feature maps from the previous layer are passed through fully connected layers (**fc1, fc2, and fc3**) to perform classification.

fc1: It has 1296 input features (corresponding to the 16x9x9 feature maps) and produces 120 output features.
fc2: It takes the 120 input features and produces 84 output features.
fc3: It takes the 84 input features and produces 10 output features, representing the class probabilities for the 10 classes.

Dropout: Dropout regularization is applied after fc1 and fc2 layers to prevent overfitting. The dropout rate is specified as a parameter during initialization.

Activation Function: The network uses an activation function (specified as 'activation') that is passed to the constructor as an argument to allow for testing with several activation functions.

Output: The final output is the result of the fc3 layer, which represents the class probabilities for the input images.

Implementation of this model in python can be seen below:

```
class Net(nn.Module):
        def __init__(self, drop=0.2):
            super().__init__()
            self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
            self.bn1 = nn.BatchNorm2d(6)
            self.pool = nn.MaxPool2d(2, 2)

            self.conv2 = nn.Conv2d(6, 16, 5)
            self.bn2 = nn.BatchNorm2d(16)

            self.fc1 = nn.Linear(16 * 9 * 9, 120)
            self.fc2 = nn.Linear(120, 84)
            self.fc3 = nn.Linear(84, 10)

            self.dropout = nn.Dropout(p=drop)

        def forward(self, x):
            x = self.pool(activation(self.bn1(self.conv1(x))))
            x = self.pool(activation(self.bn2(self.conv2(x))))
            x = torch.flatten(x, 1)
            x = activation(self.dropout(self.fc1(x)))
            x = activation(self.dropout(self.fc2(x)))
            x = self.fc3(x)
            return x
```

b. Data Structures

```
class FERPlusDataset(Dataset):
    def __init__(self, csv_file, root_dir, transform=None):
        self.img_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform

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

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()
        img_name = os.path.join(self.root_dir, self.img_frame.iloc[idx, 0])
        image = io.imread(img_name)
        image = Image.fromarray(image)
        emotions = self.img_frame.iloc[idx, 2:]
        emotions = np.asarray(emotions)
        emotions = emotions.astype('float32')
        if self.transform:
            image = self.transform(image)

        sample = {'image': image, 'emotions': emotions}

        return sample
```

The FERPlusDataset class is a custom dataset class used for loading and preprocessing data for the FERPlus dataset.

Details:

__init__(self, csv_file, root_dir, transform=None): This is the initialization method of the class. It takes three parameters:

csv_file: The path to the CSV file that contains the annotations (labels) for the dataset.

root_dir: The directory path where all the images of the dataset are located.

transform (optional): An optional parameter that represents the data transformation to be applied to the samples.
self.img_frame: This variable holds the pandas DataFrame obtained from reading the CSV file using pd.read_csv(csv_file). The DataFrame contains the annotations for each image in the dataset.

self.root_dir: This variable holds the root directory path where the images of the dataset are stored.

self.transform: This variable holds the data transformation that will be applied to each sample. It is expected to be a callable object, such as an instance of a transforms.Compose class from the torchvision.transforms module.

__len__(self): This method returns the length of the dataset, which is equal to the number of samples in the DataFrame (number of rows).

__getitem__(self, idx): This method is used to retrieve a specific sample from the dataset based on the given index idx. It performs the following steps:

- If idx is a tensor, it converts it to a Python list.
- Constructs the image file path by joining self.root_dir with the image file name from the DataFrame.
- Reads the image using io.imread() from the skimage library and converts it to a PIL Image object using Image.fromarray().
- Retrieves the emotions (labels) for the corresponding sample from the DataFrame and converts them to a NumPy array of type 'float32'.
- If a transformation is provided (self.transform is not None), it applies the transformation to the image.

- Constructs a dictionary sample with two keys: 'image' and 'emotions'. 'image' contains the transformed image, and 'emotions' contains the labels.
- Returns the sample dictionary as the final output.

Overall, this class provides an interface to load and preprocess the FERPlus dataset by reading the annotations from a CSV file, accessing the corresponding images, applying transformations if required, and returning a dictionary containing the preprocessed image and its associated labels for each sample.

c. important Methods / functions:

```
def train_and_test_untrained_CNN():
    """
    Runs a complete training and testing process for an untrained neural network model.
    hyper params are taken from "FINAL_PROJECT/utils/hyper_parameters.py" file, if you want to change any
parameters change it there.
    saves training and validation details in csv file
    create accuracy-loss plot for training and validation
    creates confusion matrix for testing
    Parameters:
    -----------
    None

    Returns:
    --------
    None
    """
    criterion, optimizer, activation_func, learning_rate, epochs, batch_size, scheduler, dropout_rate =
hps['CNN'].values()
    opt_name = optimizer.__name__
    device = get_device()
    trainloader, validloader, testloader = get_data_loaders()
    model = Net(drop=dropout_rate, activation_func=activation_func)
    model = model.to(device)
    optimizer = optimizer(model.parameters(), lr = learning_rate)
    path = create_output_directories('outputs')
    model, train_accuracy, train_loss, valid_accuracy, valid_loss, elapsed_time, scheduler_name =
train_and_validate(path=path, epochs=epochs, optimizer=optimizer, scheduler=scheduler,
criterion=criterion, model=model, trainloader=trainloader, validloader=validloader,
batch_size=batch_size, learning_rate=learning_rate, activation_func=activation_func)

    test_model(model, testloader, path)

    write_to_table(path, epochs, opt_name, criterion, batch_size, learning_rate,
activation_func.__name__, elapsed_time, train_loss, train_accuracy, valid_loss, valid_accuracy,
scheduler_name.__class__.__name__, device, dropout_rate)

    create_accuracy_loss_plot(path, epochs, train_accuracy, valid_accuracy, train_loss, valid_loss)
```

```python
def load_and_test_pretrained_CNN(pretrained_model_path):
    """
    Loads a pretrained Convolutional Neural Network model from a specified path
    and evaluates the model on the test dataset. It creates a confusion matrix.
    This function does not train the
    model.

    :param pretrained_model_path: A string specifying the path to the pretrained model.

    :return: None.
    """
    trainloader, validloader, testloader = get_data_loaders()
    model = Net()
    model.load_state_dict(torch.load(pretrained_model_path))
    print("model loaded successfully")
    model.eval()
    output_path = create_output_directories('outputs')

    test_model(model, testloader, output_path)
```

```python
def load_and_test_RESNET(pretrained_model_path):
    """
    Load a pre-trained RESNET model, perform testing, and save the results.

    Args:
        pretrained_model_path (str): Path to the pre-trained RESNET model file.

    Returns:
        None

    """
    criterion, optimizer, activation_func, learning_rate, epochs, batch_size, scheduler, dropout_rate =
hps['RESNET'].values()
    opt_name = optimizer.__name__
    device = get_device()
    trainloader, validloader, testloader = get_data_loaders(augmentation=False)
    train_dataset, validation_dataset, test_dataset = get_datasets(augmentation=False)

    model = models.resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
    num_classes = train_dataset.classes
    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, num_classes)
    model.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
    # load model from path
    model.load_state_dict(torch.load(pretrained_model_path))

    train_dataset, validation_dataset, test_dataset = get_datasets(augmentation=False)
    path = create_output_directories()

    test_model(model, testloader, path)
```

```python
def train_and_test_pretrained_pretrained_RESNET(weight_freezing=False):
    """
    Trains,validates and test a pre-trained ResNet-18 model on a custom dataset. By default, all
weights of the
    model are trainable, but by setting `weight_freezing` to True, all layers except for the first
    and last ones will have their weights frozen.

    Args:
    - weight_freezing (bool): Whether to freeze all layers except the first and last ones. Default is
False.

    Returns:
    - None
    """
    criterion, optimizer, activation_func, learning_rate, epochs, batch_size, scheduler, dropout_rate =
hps['RESNET'].values()
    opt_name = optimizer.__name__
    device = get_device()
    trainloader, validloader, testloader = get_data_loaders(augmentation=False)
    model = models.resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
    if weight_freezing:
        print("Weights of all layers in the model exept of first and last layers will be freezed")
        for param in model.parameters():
            param.requires_grad = False
    else:
        print("all weights of all layers of the models are trainable .. this might take longer")

    train_dataset, validation_dataset, test_dataset = get_datasets(augmentation=False)

    num_classes = train_dataset.classes
    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, num_classes)
    model.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
    print("model loaded successfully")
    model.to(device)
    path = create_output_directories()
    # model.load_state_dict(torch.load('../models/RESNET/RESNET-18_11.pth'))
    # print(f"model: {model}")
    model, train_accuracy, train_loss, valid_accuracy, valid_loss, elapsed_time, scheduler_name =
train_and_validate(path=path, epochs=epochs, optimizer=optimizer, scheduler=scheduler,
criterion=criterion, model=model, trainloader=trainloader, validloader=validloader,
batch_size=batch_size, learning_rate=learning_rate)

    test_model(model, testloader, path)

    write_to_table(path, epochs, opt_name, criterion, batch_size, learning_rate,
activation_func.__name__, elapsed_time, train_loss, train_accuracy, valid_loss, valid_accuracy,
scheduler_name.__class__.__name__, device, dropout_rate)

    create_accuracy_loss_plot(path, epochs, train_accuracy, valid_accuracy, train_loss, valid_loss)
```

```python
def train_and_validate(path, epochs, optimizer, scheduler ,criterion, model, trainloader, validloader,
batch_size, learning_rate, activation_func = F.relu, trial_id=0):
    """
    Trains and validates the given model using the provided data loaders and hyperparameters.

    :param path: str - Path to the directory to save the model and plots.
    :param epochs: int - Number of epochs to train the model.
    :param optimizer: function - The optimizer function to use.
    :param scheduler: function - The learning rate scheduler function to use.
    :param criterion: function - The loss function to use.
    :param model: torch.nn.Module - The model to train and validate.
    :param trainloader: torch.utils.data.DataLoader - The data loader for the training set.
    :param validloader: torch.utils.data.DataLoader - The data loader for the validation set.
    :param batch_size: int - The batch size to use for training and validation.
    :param learning_rate: float - The learning rate to use for the optimizer.
    :param activation_func: function (optional) - The activation function to use.
    :param trial_id: int (optional) - The id to use in the file name when saving the model.

    :returns: tuple - A tuple containing the trained model, training and validation accuracies and
losses, time elapsed during training, and the learning rate scheduler used.
    """
```

```
train_loss = []
    train_accuracy = []
    valid_loss = []
    valid_accuracy = []
    opt_name = optimizer.__name__
    optimizer = optimizer(model.parameters(), lr=learning_rate)

    if scheduler == optim.lr_scheduler.ReduceLROnPlateau:
        scheduler = scheduler(optimizer)
        # print("plateu")
        # print(type(scheduler))

    elif scheduler == optim.lr_scheduler.ExponentialLR:
        scheduler = scheduler(optimizer, gamma=0.9)
        # print(type(scheduler))

    st = time.time()

# Training - Validation loop
    for epoch in range(epochs):
        running_loss = 0.0
        correct = 0
        total = 0
        model.train()

        # Perform training
        for data in trainloader:
            labels = data['emotions'].to(device)
            inputs = data['image'].to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

            # Calculate and store training accuracy
            _, predicted = torch.max(outputs, 1)
            _, labels = torch.max(labels, 1)

            total += inputs.size(0)
            correct +=  (predicted == labels).sum().item()

        scheduler.step()
        train_loss.append(running_loss / len(trainloader))
        train_accuracy.append(100 * correct / total)

        # Perform validation
        model.eval()
        correct = 0
        total = 0
        running_loss = 0.0
        with torch.inference_mode():
            for data in validloader:
                labels = data['emotions'].to(device)
                inputs = data['image'].to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                running_loss += loss.item()
                _, predicted = torch.max(outputs, 1)
                _, labels = torch.max(labels, 1)
                total += inputs.size(0)
                correct +=  (predicted == labels).sum().item()

        valid_loss.append(running_loss / len(validloader))
        valid_accuracy.append(100 * correct / total)

        # Print the training and validation loss and accuracy
        print(f'Epoch {epoch+1}/{epochs}:')
        print(f'Training Loss: {train_loss[-1]:.4f} | Training Accuracy: {train_accuracy[-1]:.2f}%')
        print(f'Validation Loss: {valid_loss[-1]:.4f} | Validation Accuracy:
{valid_accuracy[-1]:.2f}%')
        print('-----------------------------------')

    elapsed_time = time.time() - st
    print('Finished Training')

    # torch.save(model.state_dict(), f'./models/RESNET/RESNET-18_{trial_id}.pth')

    return model, train_accuracy, train_loss, valid_accuracy, valid_loss, elapsed_time, scheduler
    write_to_table(path, epochs, opt_name, criterion, batch_size, learning_rate,
activation_func.__name__, elapsed_time, train_loss, train_accuracy, valid_loss, valid_accuracy,
trial_id, scheduler.__class__.__name__, device_name, dropout)

    create_accuracy_loss_plot(path, epochs, train_accuracy, valid_accuracy, train_loss, valid_loss)
```

d. User Interface / GUI

The software solution provides a graphical user interface (GUI) to interact with the facial expression recognition system. The GUI allows users to perform the following actions:

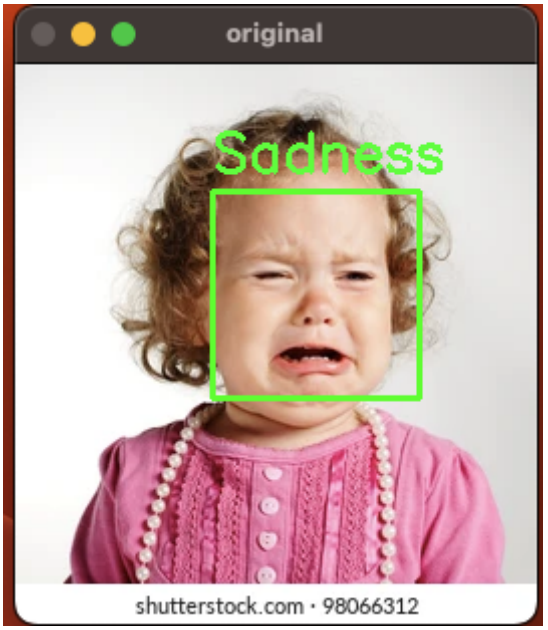
1. **Select NN**: Options are :

CNN: for choosing our own fine-tuned CNN

RESNET: to choose the RESNET-18 model that was pretrained on the ImageNet dataset and utilized transfer learning techniques to approach the facial expression classification task.

2. **Choose an image** : to choose an image file in the supported formats and classify any facial expressions if they exist.
3. **Live Classification**:

Examples:



For the GUI to detect faces before detecting the emotions, we use **OpenCV library**, which provides that out of the box.
If OpenCV couldn't identify a face, the whole image is sent to to the model.

The program utilizes **Tkinter** library to create the window and open the device's camera and file viewer.

When an NN is chosen, the corresponding trained model is loaded, then after choosing a file or opening the camera, the input is prepared to fit the models' input and passed to the model via the data loader provided from **torch.utils**. The model then returns a prediction that is then translated to a corresponding emotion category.

In the next section, we will discuss the experiments conducted, including the experimental setup, parameter variations, and result analysis. We will present the experimental background, describe the experimental results, and draw partial conclusions based on the presented experiments.

• Experiments

In this section, we will discuss the experiments conducted to evaluate the performance of the CNN-based deep learning facial expression recognition system. We will provide the experimental background, describe the experiments performed, present the results, and analyze them with partial conclusions.

a. Experimental Background

To ensure fair evaluation, we set up the experimental environment by specifying the hardware and software configurations. The experiments were conducted on a machine equipped with a high-performance Accelerated PyTorch training on Mac to accelerate the training and inference processes. The software environment included the chosen deep learning framework, PyTorch, along with the necessary dependencies and libraries.

b. Experimental Setup

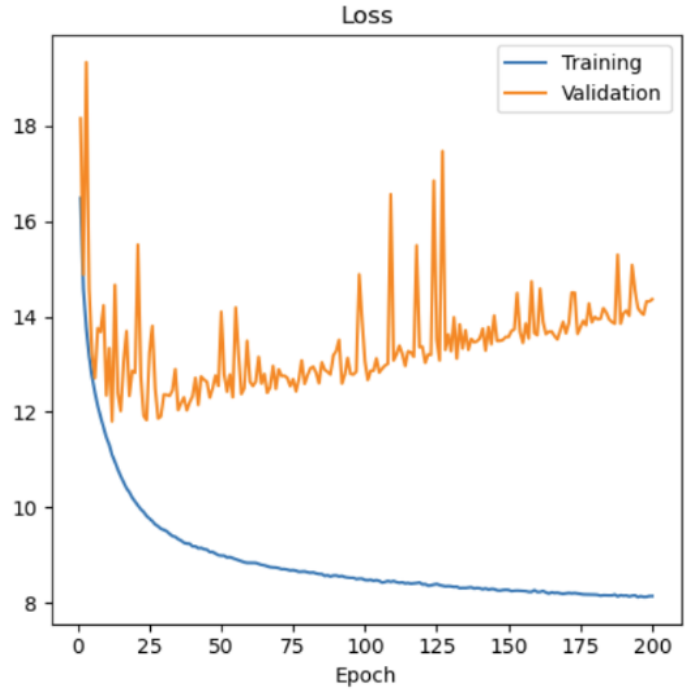
We designed a series of experiments to explore the performance of the facial expression recognition system. The experiments focused on varying specific parameters or conditions to observe their impact on the system's accuracy and robustness.

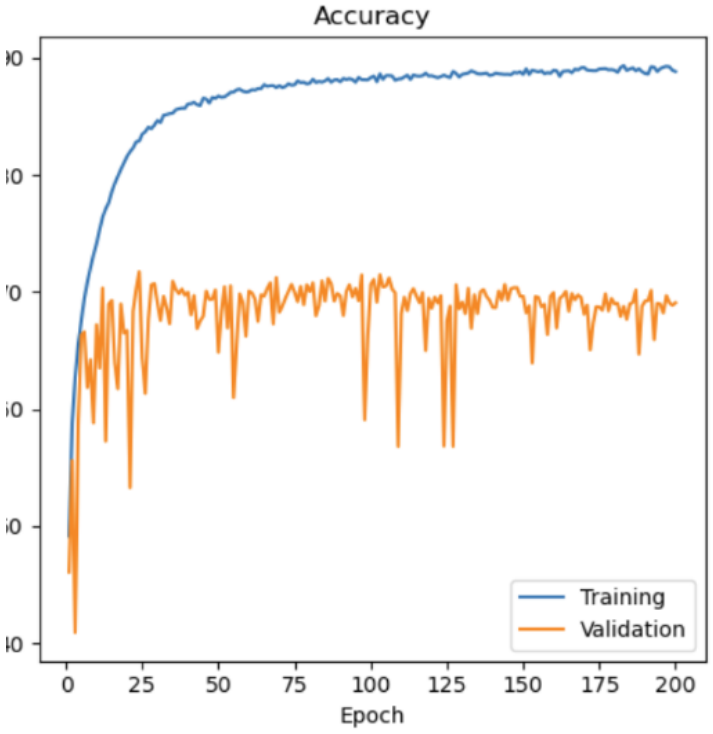
Specifically, we considered variations on the following aspects:

For the CNN:

1. Network structure:

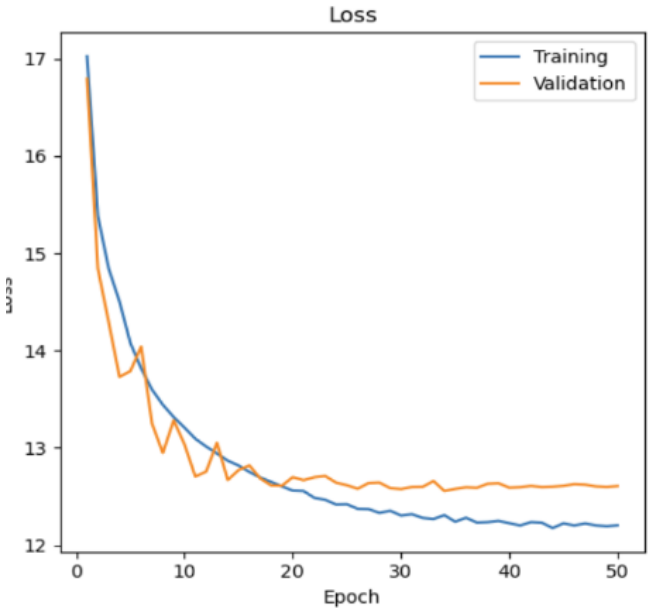
Initially, a neural network architecture was constructed with a single convolutional layer, lacking batch normalization, and comprised of only two linear layers. After training the model for 200 epochs, the maximum accuracy achieved was 69%, and evident overfitting was observed. This observation is supported by plot 1. Subsequently, the architecture was modified by adding an additional fully connected layer and convolutional layer, and implementing batch normalization. This modification significantly improved the model's performance, leading to its selection. A sample accuracy-loss graph for this chosen architecture is provided in plot 2.

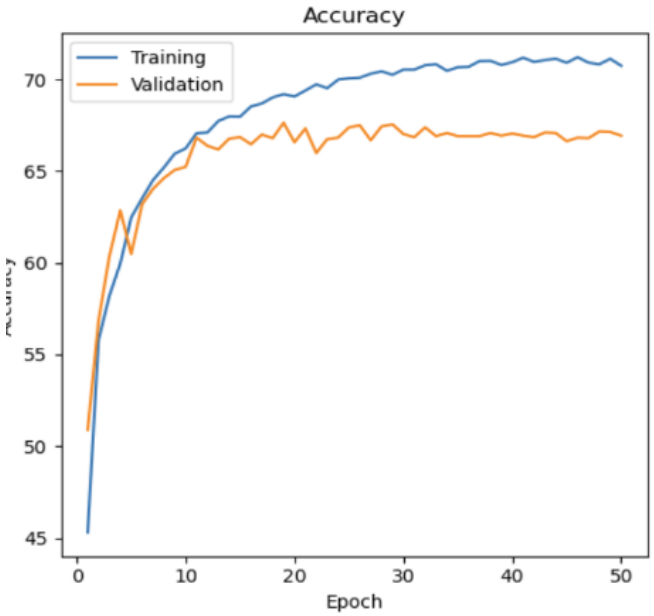




<Plot 1>

Parameter Combination:
epochs: 200
initial learning_rate: 0.01
batch_size: 64
optimizer: SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    lr: 0.01
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)
scheduler: ReduceLROnPlateau
criterion: CrossEntropyLoss()





<Plot 2>

Parameter Combination:
epochs: 50
initial learning_rate: 0.01
batch_size: 32
optimizer: Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    initial_lr: 0.01
    lr: 5.153775207320124e-05
    maximize: False
    weight_decay: 0
)
scheduler: ExponentialLR
criterion: CrossEntropyLoss()

2. Hyperparameters:

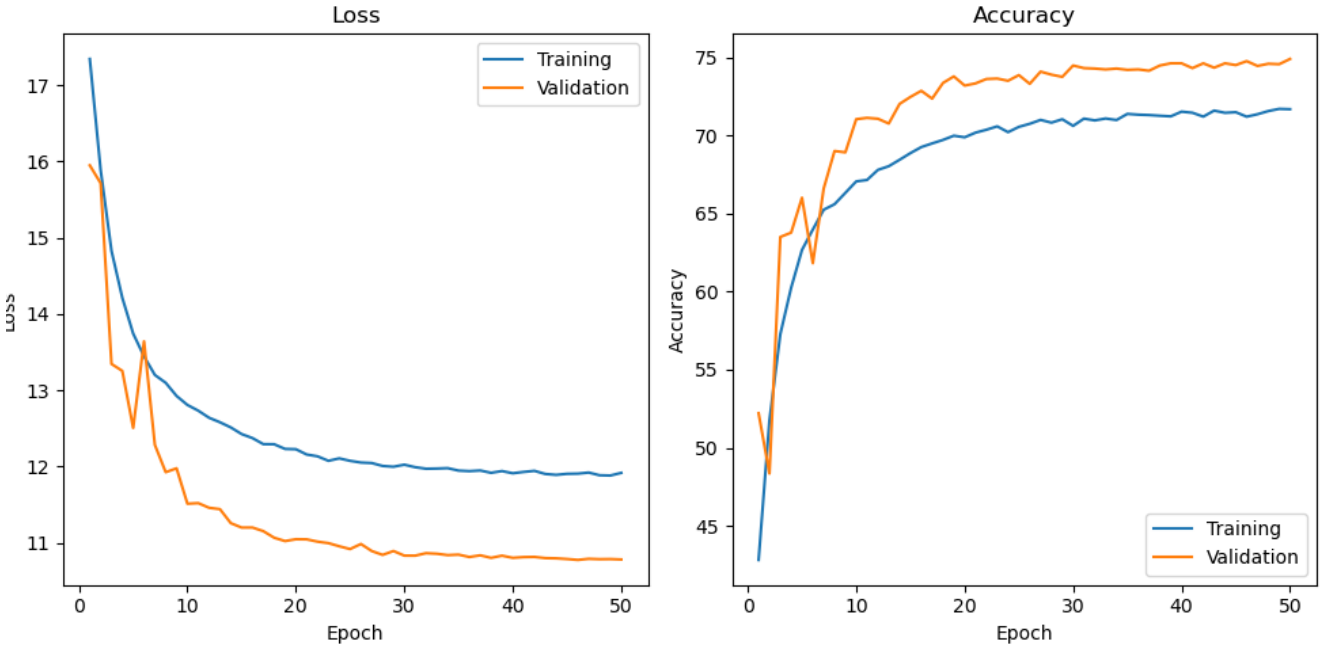
We varied hyperparameters, including:
Batch size, Epochs, Activation function, Loss function, Initial Learning rate, Optimizer, Scheduler to evaluate their effects on training convergence and model performance.
We have run over 650 experiments in total, resulting in a vast amount of statistics and plots, some of those tests were conducted at the beginning, and were faulty as a result of our inexperience. We will only show here some key samples of those tests, the ones that guided our choice of the final network hyper parameters.

loss function (Criterion):

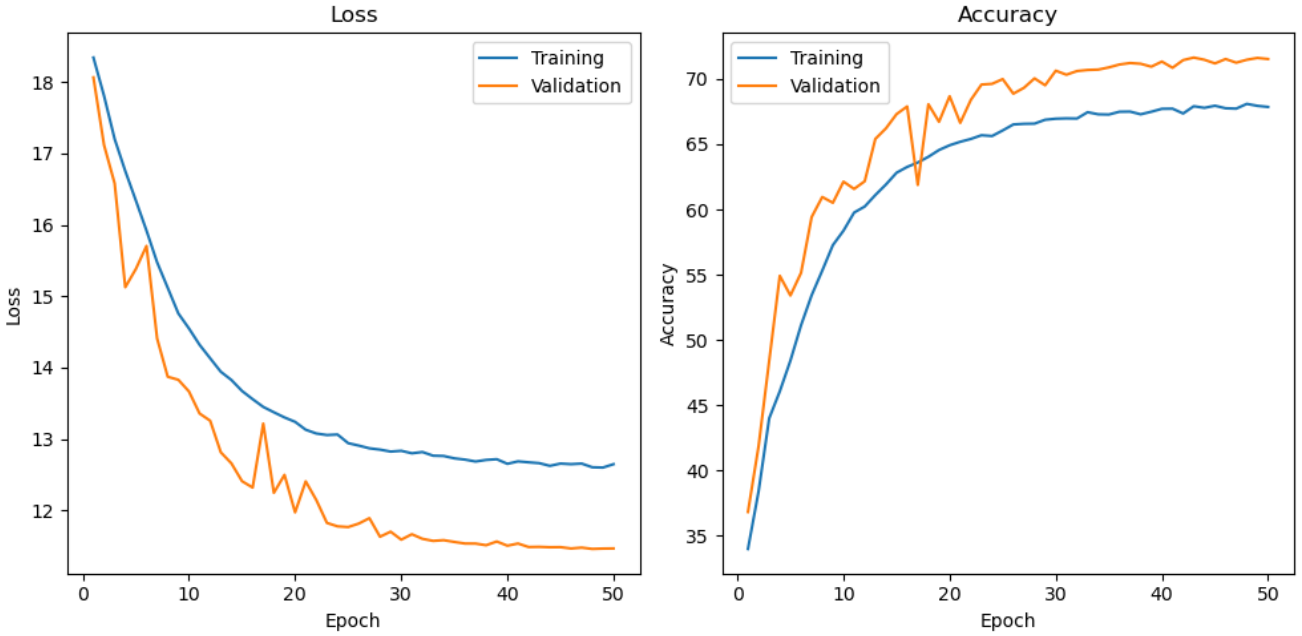
Since the task is multi class classification (and not multilabel) then **CrossEntropyLoss** is the fittest function to use.

Activation functions:

We tried **Sigmoid and ReLu**, here are some examples of the the test results:

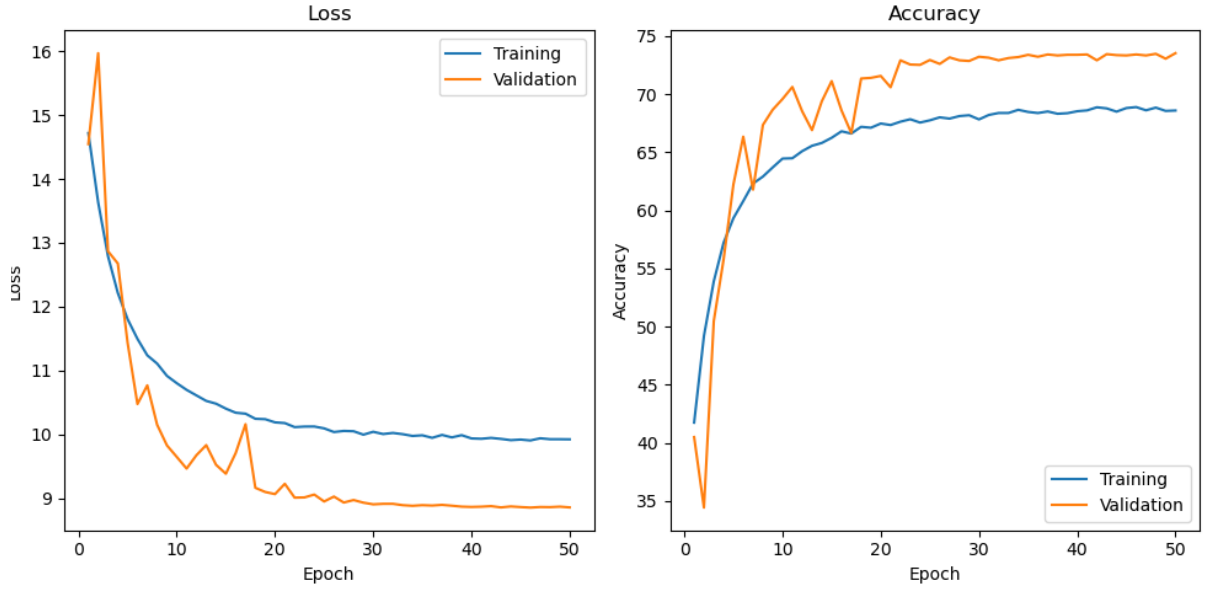


<Plot 3 >
Loss - Accuracy plot for training the CNN with **ReLu** activation function



<Plot 4>
Loss - Accuracy plot for training the CNN with **Sigmoid** activation function

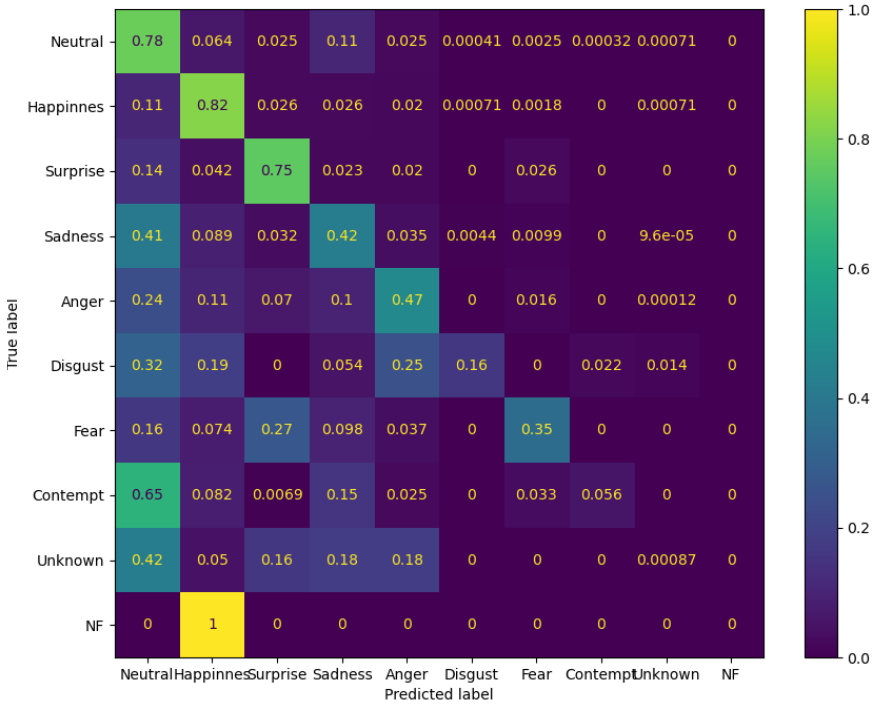
Batch Size:



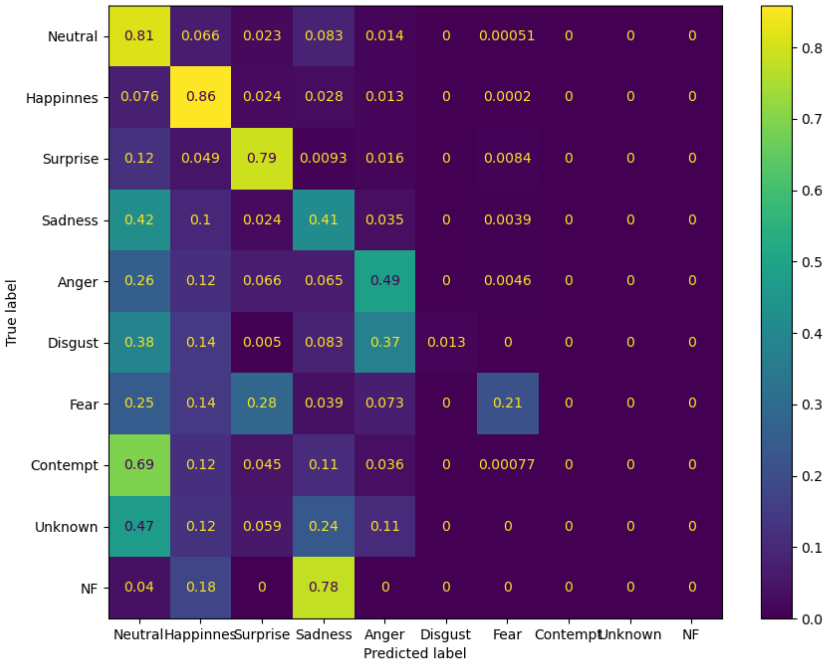
<Plot 5>
Loss - Accuracy plot for training the CNN with **Relu** activation function but with **batch size = 64**

Trials were made with different batch sizes, 6, 16, 32, 64, 128

Learning rate:

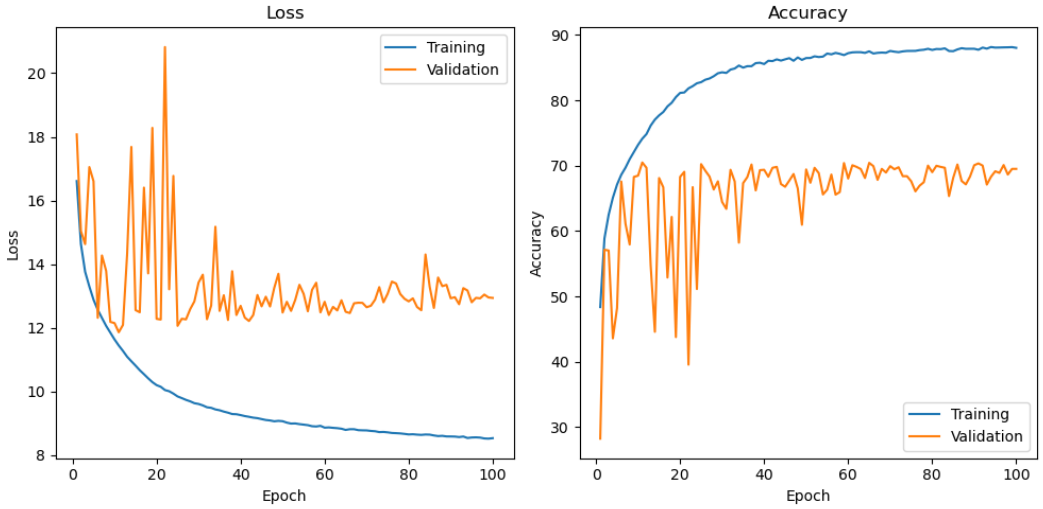


<Confusion Matrix 1>
Training details : 32 batch size , 50 Epochs, ReLu, SGD, LR = 0.001

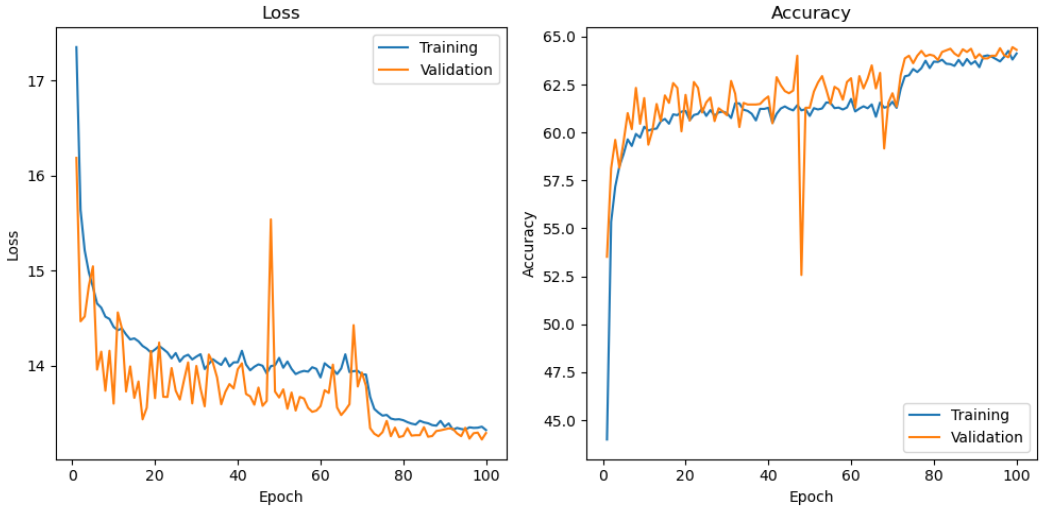


<Confusion Matrix 2>
Training details : 32 batch size , 50 Epochs, ReLu, SGD, LR = 0.01

Optimizer:

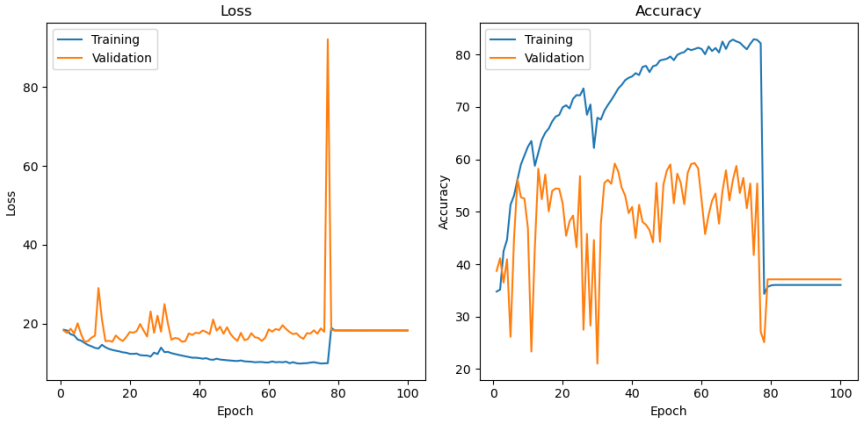


<Plot 6>
With **SGD**

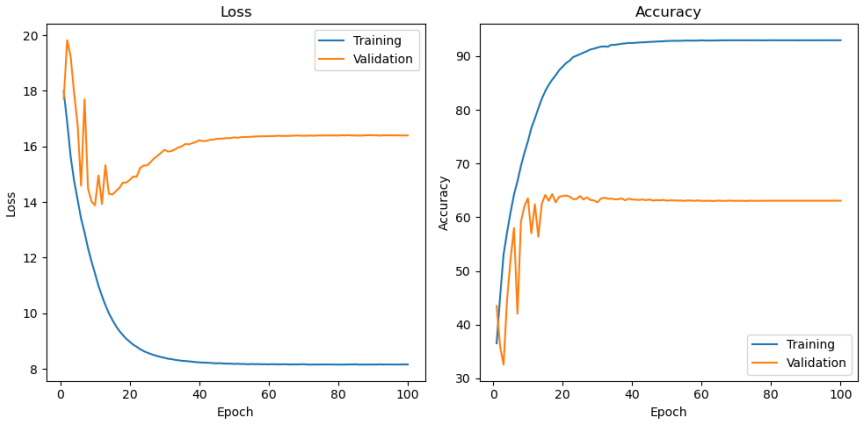


<Plot 7>
With **Adam**

Learning-rate Scheduler:



<Plot 8>
With LRS = **ReduceLROnPlateau**



<Plot 9>
With LRS = **ExponentialLR**

Dropout rates:

Changing the dropout rate between the values : 0, 0.1 and 0.2 didn't make a noticeable difference in model metrics result, but any larger than 0.2 dropped the accuracy significantly.

Data Augmentation:

we decided to use data augmentation techniques[18] to make variations to the training set as to accomplish the following goals:

- Increase Dataset Size: Data augmentation allows you to generate additional training samples by applying random transformations to the existing data. This effectively increases the size of the training dataset, which can help improve the model's performance and generalization ability.

- Reduce Overfitting: By increasing the diversity of the training data, data augmentation helps to reduce overfitting. Overfitting occurs when a model performs well on the training data but fails to generalize to unseen data. Data augmentation introduces more variations in the training samples, forcing the model to learn more robust and generalizable features.

- Improve Model Robustness: Data augmentation exposes the model to a wider range of variations and distortions that are likely to occur in real-world scenarios. By training the model on augmented data, it becomes more robust to these variations, such as changes in lighting conditions, rotations, translations, and noise.

- Balance Class Distribution: Data augmentation can be used to address class imbalance issues, where some classes in the dataset have significantly fewer samples than others.

- Learn Invariances: Data augmentation can help the model learn useful invariances or transformations that are invariant to the task at hand.

We applied the following augmentations:

RandomAffine:

Description: Randomly applies affine transformations to the image, such as rotation, translation, scaling, and shearing.
Purpose: Introduces variations in the spatial orientation and position of the image, simulating different viewpoints and perspectives.

RandomHorizontalFlip:

Description: Randomly flips the image horizontally.
Purpose: Creates variations of the image by reflecting it horizontally, which can help the model generalize to different orientations of objects.

RandomRotation:

Description: Randomly rotates the image by a certain angle.
Purpose: Introduces variations in the rotation of the image, simulating different orientations of objects and making the model more robust to object rotations.

ToTensor:

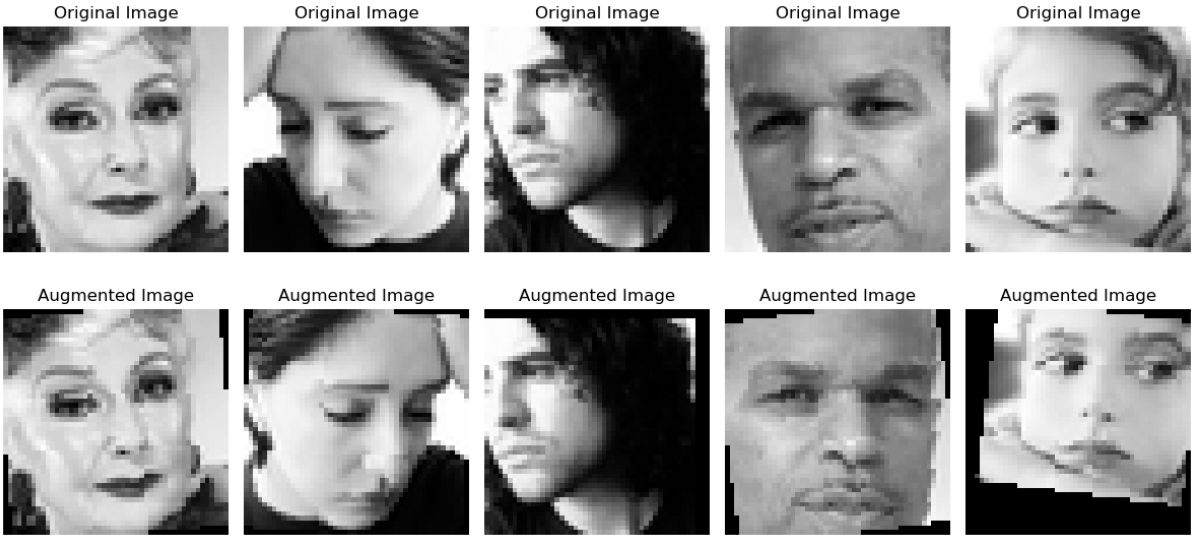
Description: Converts the image to a tensor representation.
Purpose: Transforms the image from its original format to a tensor, which is the expected input format for most deep learning models.

Normalize:

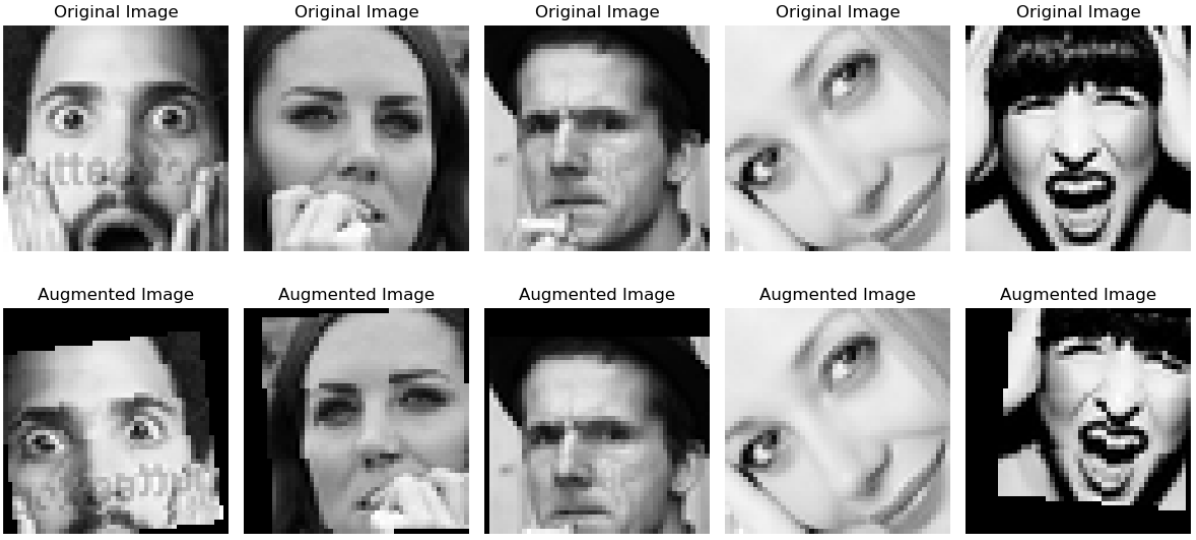
Description: Normalizes the pixel values of the image.
Purpose: Adjusts the pixel values of the image to have zero mean and unit standard deviation, which helps the model converge faster during training.

These augmentations are typically applied differently during the training and validation phases. During training, the 'train' data transformation is used, which includes random affine transformations, horizontal flipping, random rotation, converting to tensor, and normalization. These augmentations introduce variations and help the model learn from diverse perspectives. During validation, the 'val' data transformation is used, which only includes converting to tensor and normalization, ensuring a consistent representation for evaluation.

In the following images, effects of such augmentation can be seen on the training set images.



<image 1>
Data Augmentation effects



<image 2>
Data Augmentation effects

For Transfer Learning with the RESNET-18:

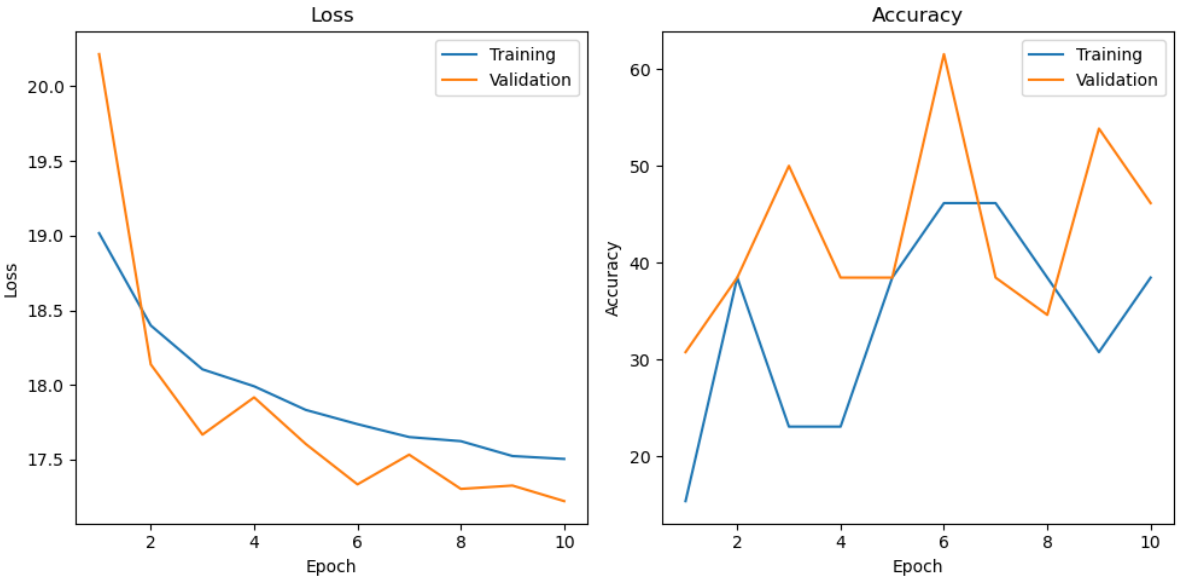
Weight Freezing[2]:

Usually there are two scenarios for using transfer learning.
These two major transfer learning scenarios look as follows:

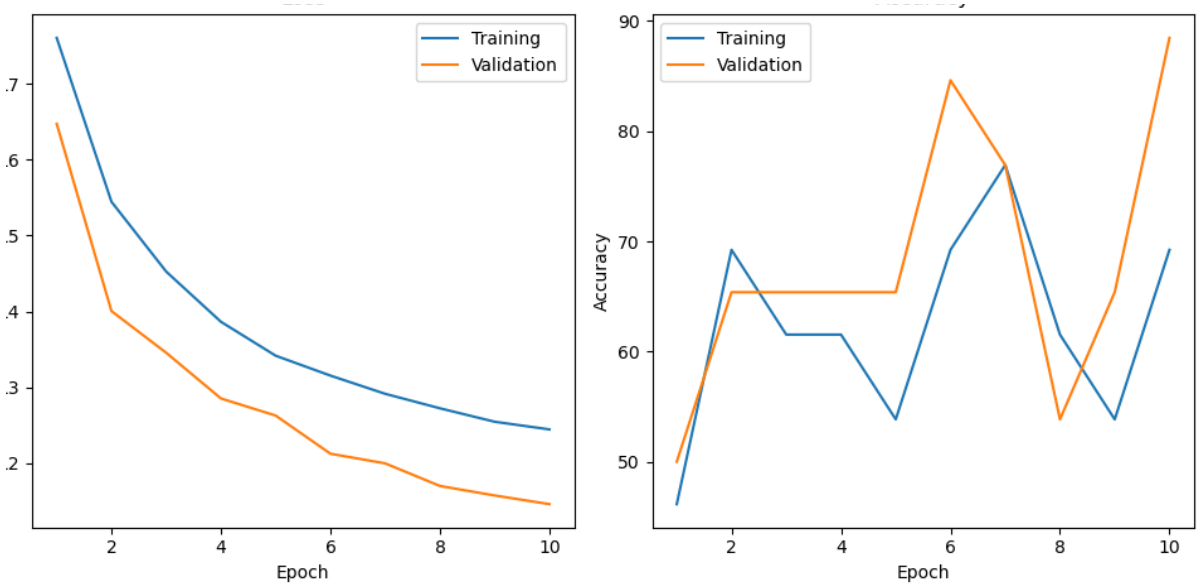
- **Fine Tuning the pretrained model:** Instead of random initialization, we initialize the network with a pretrained network, like the one that is trained on imagenet 1000 dataset. Rest of the training looks as usual.

- **Pretrained model as fixed feature extractor:** Here, we will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

We tried both approaches, Following are figures showing performance difference between the 2 methods.

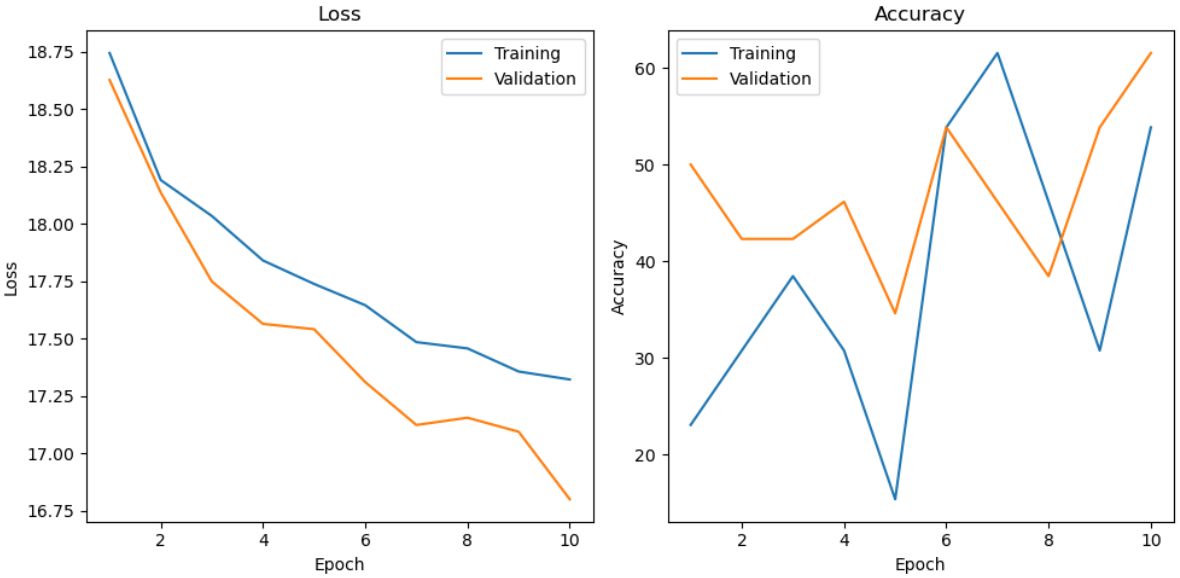


<Plot 10>
Loss-Accuracy plot for Transfer learning on the RESNET-18 model with **all weights freezed except for first and last layer**

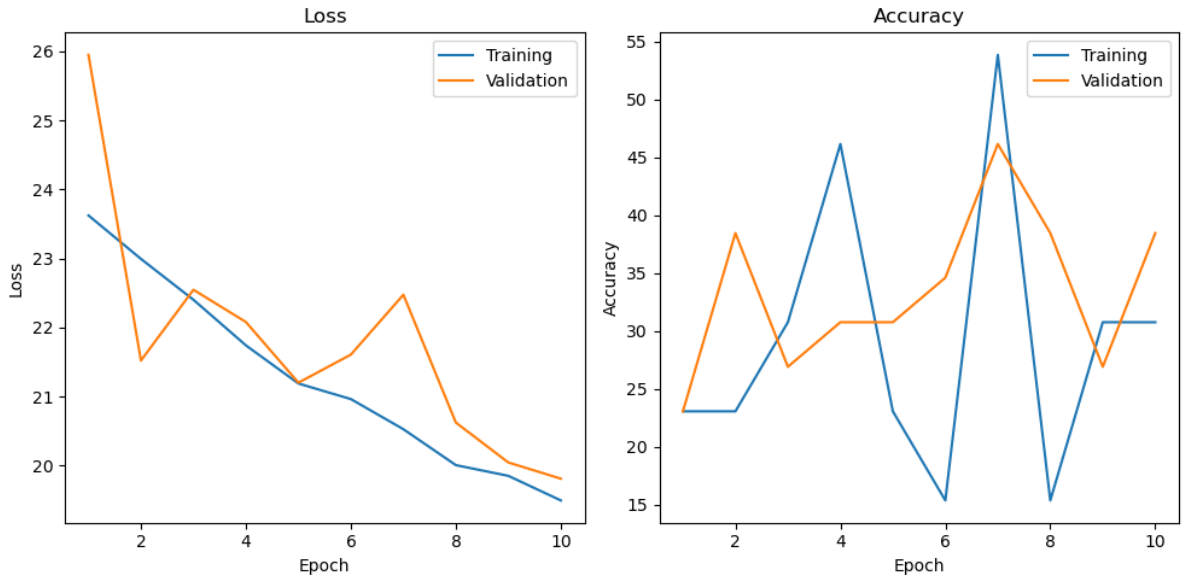


<Plot 11>
Loss-Accuracy plot for Transfer learning on the RESNET-18 model with **all weights trainable**

Learning rate:

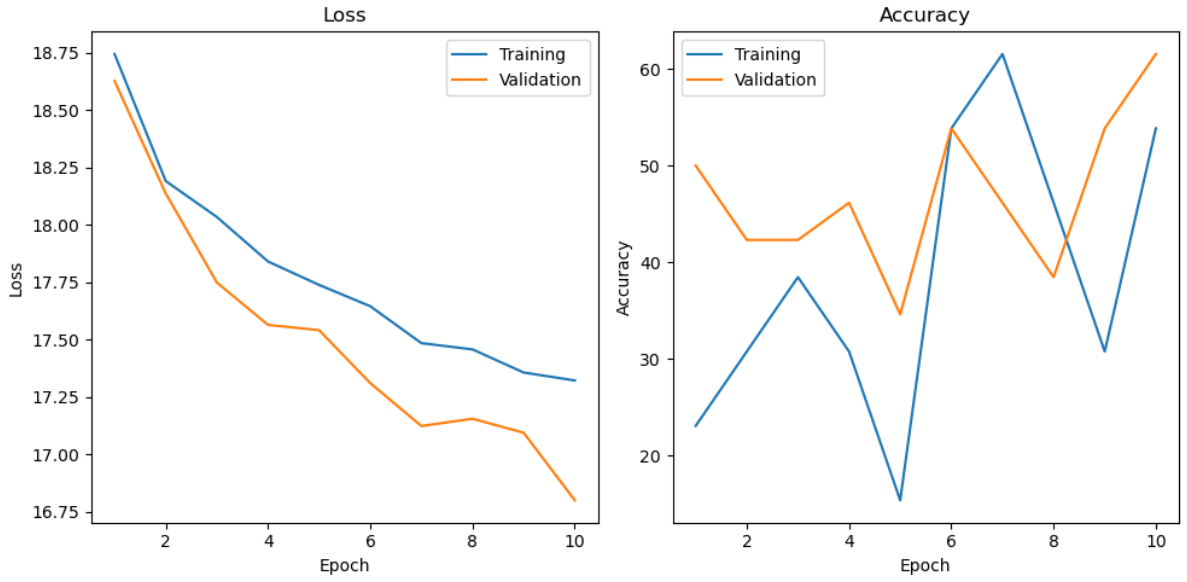


<Plot 12>
Loss-Accuracy plot for Transfer learning on the RESNET-18 model with **LR = 0.001**

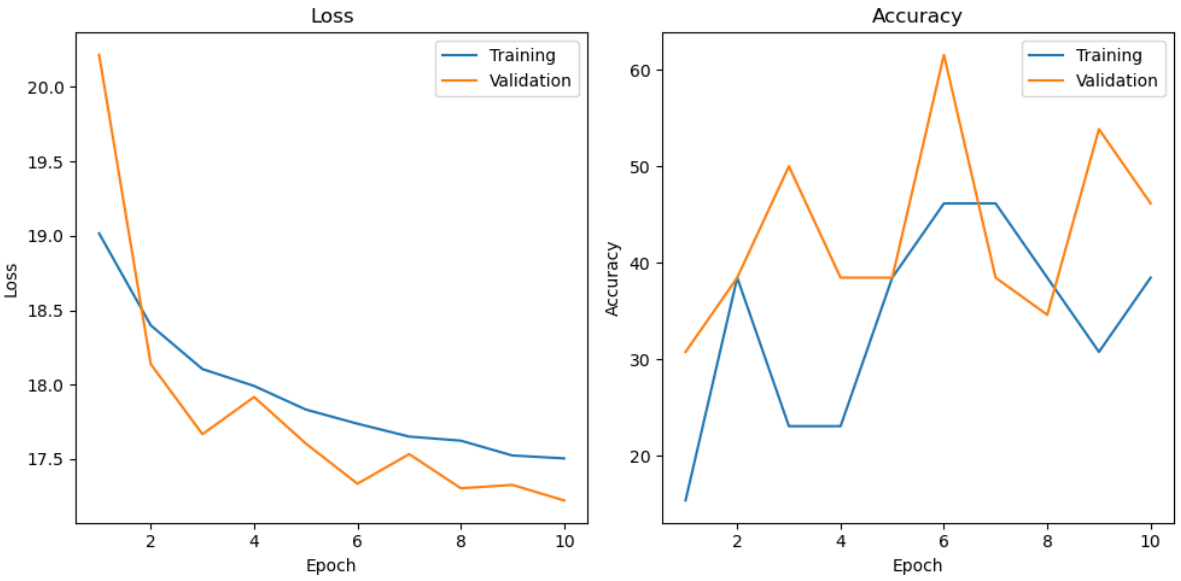


<Plot 13>
Loss-Accuracy plot for Transfer learning on the RESNET-18 model with **LR = 0.01**

Optimizer:

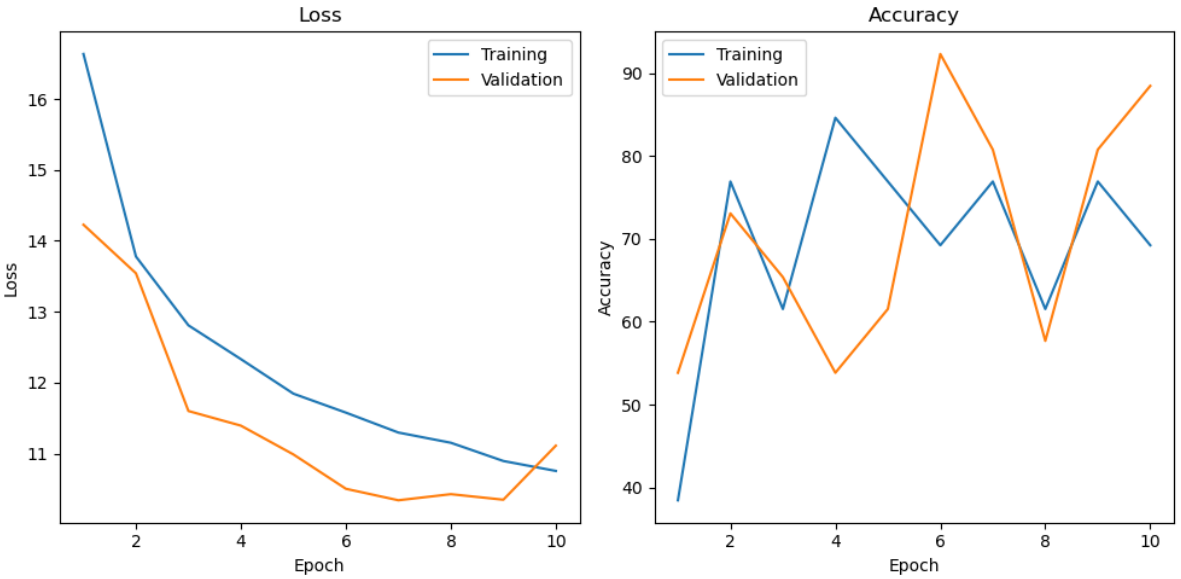


<Plot 14>
Loss-Accuracy plot for Transfer learning on the RESNET-18 model with **optim.Adam**

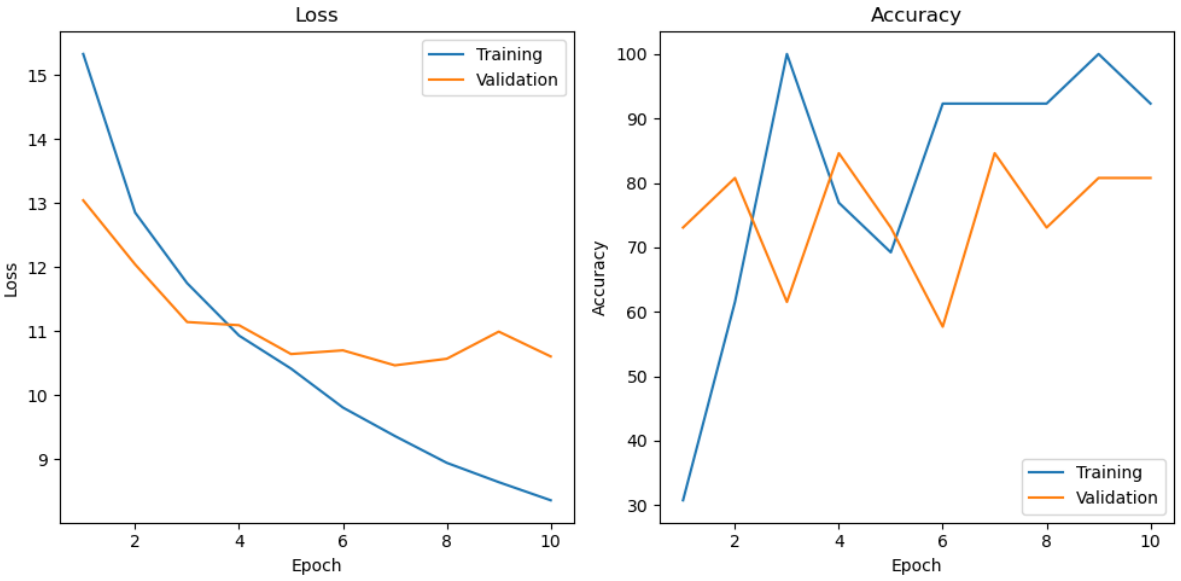


<Plot 15>
Loss-Accuracy plot for Transfer learning on the RESNET-18 model with **optim.SGD**

Data Augmentation:



<Plot 16>
Loss-Accuracy plot for Transfer learning on the RESNET-18 model **with Data Augmentation**



<Plot 17>
Loss-Accuracy plot for Transfer learning on the RESNET-18 model **without Data Augmentation**

c. Analysis & Conclusion of the hyper parameters testing

Here we come to our final conclusions drawn from the testing trials shown above.

The results are aligned with the knowledge gained from most of the references we used to better understand the task at hand.

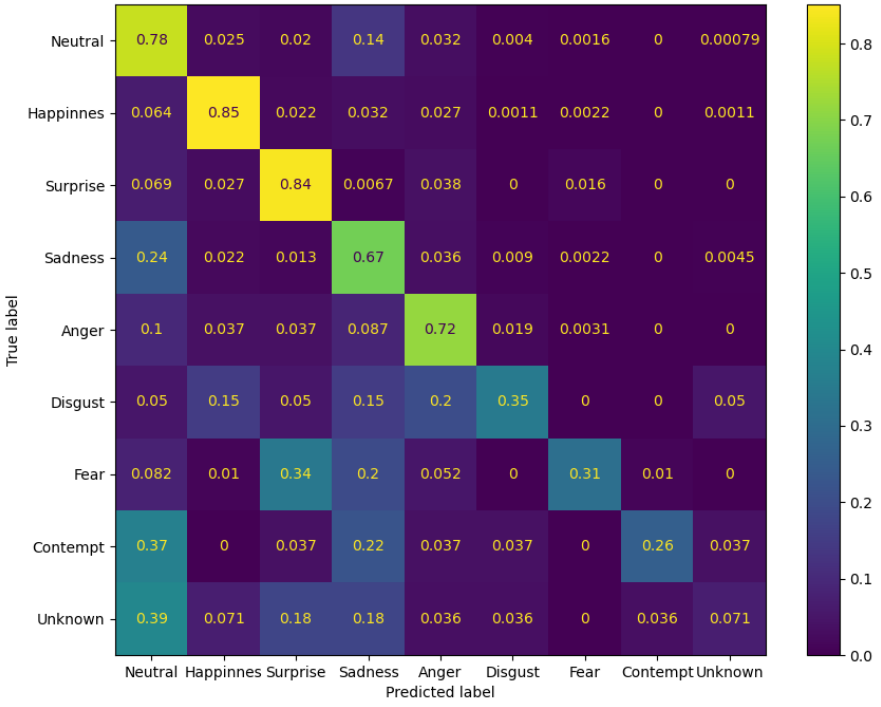
For our own CNN:

- Utilizing two convolutional layers is more effective in achieving better results compared to using only one. This is because the additional layer allows for the extraction of more complex features from the input data.
- Incorporating a batch normalization layer into the CNN model enhances its robustness. This is because batch normalization reduces internal covariate shift, which leads to faster and more stable convergence during training.
- ReLu activation function[10] performs better than Sigmoid in terms of accuracy, without any significant increase in training time. This is because ReLu is faster to compute and avoids the vanishing gradient problem associated with Sigmoid.
- A batch size of 32[11] is optimal in terms of achieving the best accuracy/time trade-off. This is because larger batch sizes can lead to overfitting, while smaller batch sizes can result in slower convergence during training. However, it was noticed that a batch size of 64 generates a slightly better confusion matrix values.
- A learning rate of 0.01 is the most effective when using a scheduler. This is because a higher learning rate can lead to unstable convergence, while a lower learning rate can result in slower convergence.
- Stochastic gradient descent (SGD) outperforms[12] Adam optimizer in terms of achieving higher accuracy. This is because SGD has better generalization performance and can escape local minima better than Adam.
- ExponentialLR scheduler is a better choice for the scheduler since it can adaptively adjust the learning rate during training based on the current loss value, leading to faster convergence, without being computationally costly[13].
- Data augmentation[14] is a useful technique in the training process, as it can increase the diversity[15] of the input data and prevent overfitting. Common data augmentation techniques include rotation, flipping, and scaling. However it was noticed that when applying data augmentation on the training set, the model performs better when dropout rate is 0 than any higher value.

For transfer learning on the RESNET-18 model:

- The RESNET-18 model shows improved performance when all weights are trainable[16], indicating that the model benefits from fine-tuning.
- Data augmentation has a negative impact on the model's performance in this case, suggesting that the model may not require additional variations in the input data.
- A lower learning rate leads to better performance for the model, indicating that a slower rate of weight updates can lead to more accurate predictions.
- The Adam optimizer is more effective for transfer learning on the RESNET-18 model, suggesting that this optimizer may be better suited for this particular task.

Here are the confusion matrices and f1-score, recall, and precision for the final trained models:



Confusion matrix of the ResNet

Evaluation Metrics:
TestAccuracy: 76.28%
F1 Score: 0.76
Precision: 0.77
Recall: 0.76

Confusion matrix of the CNN

Evaluation Metrics:
Accuracy of the network on the test images: 72.57%
F1 Score : 0.72
Precision: 0.73
Recall: 0.73

* Note : all precision, recall and F1 scores mentioned above are calculated using “weighted” average.

As it can be seen from the Metrics of the deep CNN model, test accuracy is less than the state of the art single-network accuracy which is -to our best knowledge- 73.28%.[21]
Even though we obtained test accuracy of other networks approaching 74%, we chose this one as it gave the best accuracy/confusion-matrix balance.

There is a noticeable difference in performance between the network and most other networks mentioned in scientific papers cited (for example the VGG16),

This due to the following reasons:

1. The architecture of this model is simpler than all others as explained in previous sections.
2. Almost all research was done on the FER2013 dataset, we are using the FER+. Key difference is that **FER+ contains 10 classes, not just 7**. This fact decreases the ability of the network to maintain both general accuracy and f1-score at an optimum level at the same time.
3. Because of the dataset imbalance mentioned earlier, the network becomes well-trained,in some cases even biased to classify input as one of the majority classes, and this indeed gives high accuracy.

Imagine this scenario: if the network is able to classify only “happiness” and “Neutral” with 100% precision, and fails completely with all other 8 classes, it will still have an astonishing 60.9% test accuracy even though it completely fails to identify 8 out of 10 classes.
However in the opposite case, when the model can identify the 2 major classes with lower precision and can also identify the other 8 classes with a relatively lower precision, the overall test accuracy is decreased but the model performs better in real scenarios like in the live feed classification, because it can simply identify more emotions than just 2.

● Summary

In this section, we will summarize the key findings of the project, draw overall conclusions, discuss possible improvements to the facial expression recognition system, and outline future work.

a. Summary of Findings

The project involved conducting experiments to determine the optimal hyperparameters for a convolutional neural network (CNN) and transfer learning using the RESNET-18 model. The experiments involved varying hyperparameters such as batch size, learning rate, optimizer, scheduler, activation function, and data augmentation techniques, among others. The results showed that incorporating batch normalization and using ReLu activation function, SGD optimizer, and ExponentialLR scheduler led to the best performance for the CNN model. The model achieves maximum test accuracy of 72.52% after only 100 epochs, while the state of the art CNN[20] achieves 73.2% test accuracy.

For transfer learning using the RESNET-18 model, fine-tuning all weights and using the Adam optimizer with a lower learning rate led to improved performance. Data augmentation was found to be useful for the CNN model but had a negative impact on the RESNET-18 model's performance.

b. Conclusions

Based on the results and analysis, we conclude that our facial expression recognition system performs effectively and robustly in classifying facial expressions. The high accuracy attained signifies the potential of CNNs for this task and highlights their ability to learn discriminative features from facial images.

The system shows promise in real-world applications, including emotion recognition in human-computer interaction, affective computing, and facial expression analysis for psychological research. The accurate identification of facial expressions can contribute to improving user experiences and enhancing the understanding of human emotions.

● Possible improvements & future work

a. Possible Improvements

While achieving a high accuracy of 71.6%, there are still areas for potential improvement in the facial expression recognition system. Some possible avenues for enhancement include:

Dataset Expansion: Increasing the size and diversity of the training dataset could improve the model's ability to generalize to unseen facial expressions and variations.

Fine-tuning Model Hyperparameters: Further exploration and optimization of model hyperparameters, such as learning rate, batch size, and dropout rate, may lead to improved performance.

Architectural Enhancements: Experimenting with more complex CNN architectures.

Handling Imbalanced Data: applying techniques like oversampling, undersampling, or class weighting can help address the issue and improve the system's performance on minority classes.

Exploring genetic programming and evolutionary algorithms.

Utilizing Regularization techniques.

b. Future Work

The facial expression recognition system can be extended and improved in various ways. Some avenues for future work include:

Integration in other systems, for example recommendation systems for movies and/or E-books.

Multi-modal Fusion: Exploring the integration of multiple modalities, such as combining facial expression analysis with voice or body language recognition, to enhance the overall understanding of human emotions.

**Privacy and Ethical Considerations: Addressing privacy concerns by incorporating techniques that ensure the anonymity and security of individuals' facial data during the recognition process.*

By pursuing these improvements and future work, the facial expression recognition system can advance further and find applications in various domains that require accurate emotion analysis.

Finally, in the last section, we will provide a list of references used throughout the project, acknowledging the sources that have contributed to the development and understanding of the facial expression recognition system.

- **References**

[1] Huang, ZY., Chiang, CC., Chen, JH. et al. A study on computer vision for facial emotion recognition. Sci Rep 13, 8425 (2023). https://doi.org/10.1038/s41598-023-35446-4

[2] Lin, S., Tseng, Y., Wu, C., Kung, Y., Chen, Y., & Wu, C. (2019). A Continuous Facial Expression Recognition Model based on Deep Learning Method. 2019 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS). https://doi.org/10.1109/ISPACS48206.2019.8986360.

[3] Challenges in Representation Learning: Facial Expression Recognition Challenge

[4] Microsoft - Ferplus

[5] Rekha, G., Tyagi, A., & Reddy, V. (2018). A Novel Approach to Solve Class Imbalance Problem Using Noise Filter Method. . https://doi.org/10.1007/978-3-030-16657-1_45

[6] Yan, Y., Liu, R., Ding, Z., Du, X., Chen, J., & Zhang, Y. (2019). A Parameter-Free Cleaning Method for SMOTE in Imbalanced Classification. IEEE Access. https://doi.org/10.1109/ACCESS.2019.2899467.

[7] Thakkar, V., Tewary, S., & Chakraborty, C. (2018). Batch Normalization in Convolutional Neural Networks — A comparative study with CIFAR-10 data. 2018 Fifth International Conference on Emerging Applications of Information Technology (EAIT). https://doi.org/10.1109/EAIT.2018.8470438.

[8] Kulkarni, M., Kakad, S., Mehra, R., & Mehta, B. (2020). Camera Model Identification Using Transfer Learning. . https://doi.org/10.1007/978-981-15-1884-3_6.

[9] Bera, S., & Shrivastava, V. (2020). Effect of pooling strategy on convolutional neural network for classification of hyperspectral remote sensing images. IET Image Process.. https://doi.org/10.1049/iet-ipr.2019.0561.

[10] Li, Y., Ye, X., & Li, Y. (2017). Image quality assessment using deep convolutional networks. AIP Advances. https://doi.org/10.1063/1.5010804.

[11] Lin, R. (2022). Analysis on the Selection of the Appropriate Batch Size in CNN Neural Network. 2022 International Conference on Machine Learning and Knowledge Engineering (MLKE). https://doi.org/10.1109/MLKE55170.2022.00026.

[12] Djamal, E., Ramadhan, R., Mandasari, M., & Djajasasmita, D. (2020). Identification of post-stroke EEG signal using wavelet and convolutional neural networks. Bulletin of Electrical Engineering and Informatics. https://doi.org/10.11591/EEI.V9I5.2005.

[13] Onyema, E., Shukla, P., Dalal, S., Mathur, M., Zakariah, M., & Tiwari, B. (2021). Enhancement of Patient Facial Recognition through Deep Learning Algorithm: ConvNet. Journal of Healthcare Engineering. https://doi.org/10.1155/2021/5196000.

[14] Patel, M., Wang, X., & Mao, S. (2020). Data augmentation with conditional GAN for automatic modulation classification. Proceedings of the 2nd ACM Workshop on Wireless Security and Machine Learning. https://doi.org/10.1145/3395352.3402622.

[15] Barbosa, J., Seo, W., & Kang, J. (2019). paraFaceTest: an ensemble of regression tree-based facial features extraction for efficient facial paralysis classification. BMC Medical Imaging. https://doi.org/10.1186/s12880-019-0330-8.

[16] Kruithof, M., Bouma, H., Fischer, N., & Schutte, K. (2016). Object recognition using deep convolutional neural networks with complete transfer and partial frozen layers. . https://doi.org/10.1117/12.2241177.

[17] Jiang, F., Liu, H., Yu, S., & Xie, Y. (2017). Breast mass lesion classification in mammograms by transfer learning. Proceedings of the 5th International Conference on Bioinformatics and Computational Biology. https://doi.org/10.1145/3035012.3035022.

[18] Luo, J., Xie, Z., Zhu, F., & Zhu, X. (2021). Facial Expression Recognition using Machine Learning models in FER2013. 2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer (ICFTIC). https://doi.org/10.1109/ICFTIC54370.2021.9647334.

[19] Wu, Y., Zhang, L., Chen, G., & Michelini, P. (2021). Unconstrained Facial Expression Recogniton Based on Cascade Decision and Gabor Filters. 2020

25th International Conference on Pattern Recognition (ICPR). https://doi.org/10.1109/ICPR48806.2021.9411983.

[20] Pramerdorfer, C., & Kampel, M. (2016). Facial Expression Recognition using Convolutional Neural Networks: State of the Art. ArXiv.

[21] Khaireddin, Y., & Chen, Z. (2021). Facial Emotion Recognition: State of the Art Performance on FER2013. ArXiv.

- **links**

Drive:https://drive.google.com/drive/folders/1CP9oaZs20bJ9WuRUK2CYq6_tumX3tVPQ?usp=sharing

Github Repo : https://github.com/AsserElfeki/FER_PLUS-CNN