**Project topic**: Face Expression Automated Recognition (FEAR)

**Team:** Egorova Yana (frontend), Lubetskaya Antonina (ML), Kastuba Danila (backend), Smirnova Julia (project manager)

**INTRODUCTION**

Facial expression recognition (FER) is a crucial task within the fields of computer vision, human-computer interaction, and affective computing. It involves automatically identifying and classifying human emotions based on facial expressions. Emotions such as happiness, sadness, anger, surprise, fear, and disgust are typically conveyed through subtle changes in facial muscles, which are often challenging for machines to detect accurately. The ability to recognize these expressions is essential for applications ranging from personalized user experiences to mental health monitoring and security systems. Human emotions are expressed in diverse ways influenced by factors such as age, gender, ethnicity, and cultural background. A single emotion like “happiness” can manifest differently across individuals, making it difficult to build generalized models. Many facial expressions are subtle or ambiguous. For example, distinguishing between "fear" and "surprise" or between "neutral" and "sad" requires high precision, as these expressions may only differ by minor muscle movements. This is a challenging task for machine learning models, as it requires a deep understanding of human emotions and their subtle differences.

**PROBLEM STATEMENT**

Most AI works perform reasonably well on datasets of images captured in a controlled condition, but fail to perform as good on more challenging datasets with more image variation and partial faces. In recent years, several works proposed an end-to-end framework for facial expression recognition, using deep learning models. Despite the better performance of these works, there still seems to be a great room for improvement. So we need to make a web application, which will recognize different emotions.

To achieve this goal, some specific tasks have been identified, namely:

* find a dataset with different emotion options;
* train models Decision Tree, ViT;
* evaluate models based on test data;
* compare models with each other;
* design a user-friendly web interface using HTML and CSS;
* implement the backend using the Django framework;
* store and manage data in PostgreSQL database;
* incorporate a selected machine learning model into the system.

**DATASET**

So, to train the model to recognize emotions in people's faces, we needed to select a suitable dataset. We stopped at FER2013, as there are about 30,000 images in the dataset, and each image contains one of the basic emotions: Angry, Disgust, Fear, Happy, Sad, Surprise, or Neutral.

**MODEL BUILDING STAGE**

We have selected two models for our system: traditional machine learning techniques (Decision Tree Classifier) and state of the art deep learning models (Vision Transformer). We will look at each model individually before comparing them.

**Decision Tree Classifier**

This is a model that makes decisions based on a series of rules arranged in a tree structure. Each node in the tree represents a test on an attribute (also known as a feature), and the branches represent the results of that test leading to other nodes or a final decision (or class).

The main purpose of the algorithm is to create a model that can classify input data by making decisions at each step of the tree, based on the values of the attributes.

Let us examine how this method functions in our code.

1. Extracting features for training

X\_train, y\_train = extract\_features(train\_loader, device)

X\_test, y\_test = extract\_features(test\_loader, device)

The attributes and class labels for the training and test samples are extracted. The extract\_features() function is used to process the data and obtain the appropriate input data for training the model. This data is divided into two arrays:

1. X\_train, X\_test – feature sets (numerical data extracted from images);
2. y\_train, y\_test are class labels (labels indicating emotions or categories).
3. Scaling of features

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

Feature scaling is important to improve model performance. This code uses StandardScaler to normalize the data:

1. fit\_transform(X\_train) calculates the mean and standard deviation for the training data and applies the transformation;
2. transform(X\_test) applies the same scaler to the test data using the same parameters obtained from the training data.
3. Learning the decision tree model

dt\_model = DecisionTreeClassifier(random\_state=42)

dt\_model.fit(X\_train\_scaled, y\_train)

Here, a DecisionTreeClassifier model is created and trained using the fit() method. The model is initialized with the parameter random\_state=42 to ensure reproducibility of the results.Decision tree training: fit method(X\_train\_scaled, y\_train) trains the model on the training data:

1. X\_train\_scaled are the attributes that will be used to make decisions.
2. y\_train — class labels (emotions, categories, etc.) that correspond to these attributes.

During training, the algorithm will look for optimal features to split the data into subsets to minimize error.

For each node of the tree, the algorithm selects the best feature for data separation. Partitioning continues recursively until the stopping criteria are reached, such as the maximum depth of the tree or the minimum amount of data to split.

1. Predictive analytics based on test data

y\_pred\_dt = dt\_model.predict(X\_test\_scaled)

After training the model, prediction is performed on the test data using the predict() method, which returns class labels for the test data using a trained decision tree model: X\_test\_scaled — test features (scaled)

1. Evaluating the accuracy of the model

dt\_accuracy = accuracy\_score(y\_test, y\_pred\_dt)

print(f"Decision Tree Accuracy: {dt\_accuracy \* 100:.2f}%")

After the model has made predictions, the accuracy is evaluated using the accuracy metric, which is calculated as the proportion of correct predictions from the total number of test cases:

1. y\_test are the true class labels for the test data.
2. y\_pred\_dt are predicted class labels by the model.
3. saving the trained model and scaler

import joblib

# Сохранение модели Decision Tree

joblib.dump(dt\_model, 'decision\_tree\_model.pkl')

print("Decision Tree model saved!")

# Сохранение скейлера

joblib.dump(scaler, 'scaler.pkl')

print("Scaler model saved!")

Here, the trained model and the scaler are saved to files. The decision tree model is saved using joblib.dump(), which allows you to load the model later without re-training. The scaler is saved so that the same scaling parameters can be used for new data when loading the model.

**Vision Transformer**

ViT ​​is a neural network architecture based on the principles of transformers, which were originally developed for processing sequences of data such as text. ViT applies these principles to computer vision tasks such as image classification.

Let us examine how this method functions in our code.

1. Loading a pre-trained ViT model

vit\_model = ViTForImageClassification.from\_pretrained("google/vit-base-patch16-224-in21k", num\_labels=7)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

vit\_model.to(device)

Model loading: A pre-trained ViT model is used here, which is trained on a large ImageNet-21k dataset using the from\_pretrained method. The model is loaded with parameters:

* + "google/vit-base-patch16-224-in21k" is a model identifier indicating that the ViT base model is loaded with 16x16 patches and 224x224 image size, trained on 21,000 classes from the ImageNet dataset.
  + num\_labels=7 — this parameter indicates that the number of classes for your classification task is 7. This is necessary to set up the last classification layer of the model, which will have 7 outputs (corresponding to the number of classes in the task).

Transferring the model to the GPU or CPU: If a graphics card (GPU) is available, the model will be transferred to the GPU, otherwise to the CPU.

1. Defining the optimizer

optimizer = torch.optim.AdamW(vit\_model.parameters(), lr=1e-5)

The AdamW optimizer is being created here to train the model. The optimizer will update the model parameters (including weights and offsets) during training using the Adam algorithm with Weight Decay. lr=1e-5 is the learning rate, which controls the step size by which the optimizer will update the model parameters at each iteration.

1. ViT Training Function

def train\_vit(model, train\_loader, optimizer, epochs=5):

    model.train()

    total\_loss = 0

    correct\_predictions = 0

    total\_predictions = 0

    loss\_fn = torch.nn.CrossEntropyLoss()

    for epoch in range(epochs):

        for images, labels in train\_loader:

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

            optimizer.zero\_grad()

            outputs = model(images).logits

            loss = loss\_fn(outputs, labels)

            loss.backward()

            optimizer.step()

            total\_loss += loss.item()

            \_, preds = torch.max(outputs, 1)

            correct\_predictions += torch.sum(preds == labels).item()

            total\_predictions += labels.size(0)

        accuracy = 100 \* correct\_predictions / total\_predictions

        print(f"Epoch {epoch + 1}/{epochs} - Loss: {total\_loss:.4f} - Accuracy: {accuracy:.2f}%")

Analyzing the operation of the train\_vit function:

1. model.train() puts the model into training mode. This is important because some layers, such as Dropout or BatchNorm, behave differently during training and inference.
2. Initialization of variables:
   * total\_loss = 0 — to accumulate the total loss per epoch.
   * correct\_predictions = 0 — to count the number of correct predictions.
   * total\_predictions = 0 — to count the total number of predictions.
3. Objective function: loss\_fn = torch.nn.CrossEntropyLoss() — the CrossEntropyLoss loss function is used, which is suitable for multiclass classification.
4. Within the epoch cycle , the model is trained on training data:
   * for images, labels in train\_loader: — a loop through data bundles from train\_loader, where images are the images in the current batch and labels — true class labels.
   * images, labels = images.to (device), labels.to (device) — data and labels are moved to the device (GPU or CPU).
5. Updating the model parameters:
   * optimizer.zero\_grad() — the gradient for all parameters is reset before calculating a new gradient.
   * outputs = model(images).logits — transfer of images through the model. We get logits (immediate predictions before using softmax) for classification.
   * loss = loss\_fn(outputs, labels) — calculation of the loss (error) between predictions and true labels using the loss function.
   * loss.backward() is the reverse propagation of the error, that is, the calculation of gradients.
   * optimizer.step() — updating the model parameters using the optimizer.
6. Calculating accuracy:
   * \_, preds = torch.max(outputs, 1) — getting predicted classes. torch.max selects the index of the maximum value in the logs, which corresponds to the predicted class.
   * correct\_predictions += torch.sum(preds == labels).item() — adding the number of correct predictions for this batch.
   * total\_predictions += labels.size(0) — adding the total number of predictions for this batch.
7. Accuracy is calculated after each epoch as a percentage of correct predictions out of the total:

accuracy = 100 \* correct\_predictions / total\_predictions

And it is displayed on the screen:

print(f"Epoch {epoch + 1}/{epochs} - Loss: {total\_loss:.4f} - Accuracy: {accuracy:.2f}%")

1. Evaluation of the ViT model

def evaluate\_vit(model, test\_loader):

    model.eval()

    correct\_predictions = 0

    total\_predictions = 0

    with torch.no\_grad():

        for images, labels in test\_loader:

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

            outputs = model(images).logits

            \_, preds = torch.max(outputs, 1)

            correct\_predictions += torch.sum(preds == labels).item()

            total\_predictions += labels.size(0)

    accuracy = 100 \* correct\_predictions / total\_predictions

    print(f"ViT Test Accuracy: {accuracy:.2f}%")

Analyzing the operation of the evaluate\_vit function:

1. model.eval() puts the model into evaluation mode. In this mode, the model behaves as in the inference.
2. Calculating accuracy on test data:
   * The evaluate\_vit function evaluates the accuracy of the model based on the test data.
   * with torch.no\_grad(): — a block of code in which gradients are not calculated, which speeds up the inference process.
   * The test data cycle is similar to the training cycle, but without updating the model weights.
   * The accuracy is calculated on the test dataset and the result is displayed.

5. Saving the ViT model

torch.save(vit\_model.state\_dict(), 'vit\_model.pth')

print("ViT model saved!")

After the training is completed, the model is saved to a file using torch.save(). Only the model parameters (weights) are saved, not the entire model itself. This allows you to download the model later and use it for predictions without re-training.

**COMPARE MODELS**

In this section, we will compare two emotion classification models: a **Decision Tree** model and a **Vision Transformer (ViT)** model. The comparison will focus on their performance in terms of prediction accuracy, computational cost, and practical applicability.

**1. Decision Tree Model**

A Decision Tree is a simple, interpretable model that makes decisions by splitting the data based on feature values. Each internal node represents a condition on an attribute, and branches represent outcomes leading to either a class label or another decision node.

**Advantages:**

* **Simplicity and interpretability:** The decision-making process in a Decision Tree can be visualized, making it easy to understand and interpret.
* **Fast training time:** Decision Trees generally train faster compared to more complex models such as neural networks.
* **No need for data normalization:** Decision Trees can work with raw, unnormalized data.

**Disadvantages:**

* **Overfitting:** Decision Trees are prone to overfitting, especially when trained on small datasets with many features.
* **Limited performance on complex patterns:** They often struggle with highly non-linear or complex patterns, which can impact their effectiveness on more intricate image-based tasks like emotion classification.
* **Instability:** Small changes in the dataset can result in a completely different tree structure.

**2. Vision Transformer (ViT) Model**

The Vision Transformer (ViT) is a deep learning architecture based on transformer models, which have gained significant popularity in natural language processing (NLP). ViT splits an image into patches and treats these patches similarly to tokens in NLP transformers, capturing spatial relationships in the process.

**Advantages:**

* **State-of-the-art performance:** ViT has demonstrated impressive results, particularly in tasks involving images. By leveraging self-attention mechanisms, it can capture complex relationships in visual data.
* **Scalability:** Vision Transformers perform well with larger datasets and can improve significantly when more data is available.
* **Better handling of complex patterns:** The self-attention mechanism allows the model to capture complex relationships and fine-grained patterns, making it ideal for tasks like emotion detection where subtle features may be significant.

**Disadvantages:**

* **High computational cost:** Training ViT models can be resource-intensive, requiring significant GPU and memory resources.
* **Longer training time:** Due to the complexity of the model, the training time for ViT is longer compared to simpler models like Decision Trees.
* **Data dependency:** ViT models usually require large datasets to avoid overfitting and achieve optimal performance.

**Comparison of Performance:**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Decision Tree** | **Vision Transformer** |
| **Accuracy** | Lower, less effective on complex patterns | Higher accuracy, especially on complex data |
| **Training Time** | Fast to train | Much slower due to complexity |
| **Interpretability** | High, easy to understand and visualize | Low, considered a "black-box" model |
| **Computational Cost** | Low memory usage, less computational power | High memory and computational resources required |
| **Overfitting Tendency** | High if not pruned properly | Less prone to overfitting with enough data |
| **Application** | Better for small datasets or simple tasks | Best for large, complex datasets with rich patterns |

**Conclusion**

In this comparison, **Decision Trees** are more suited for simpler, smaller datasets where interpretability is key, and computational resources are limited. On the other hand, **Vision Transformers** excel at handling complex patterns in large image datasets and deliver high accuracy but require substantial computational power and time to train. The choice of model depends largely on the specific use case, the dataset at hand, and available resources.

The next section will cover the final evaluation of these models and discuss the model that performs best for real-time emotion recognition in images.