Face Mask Detection, Classification, and Segmentation

Aryan Singhal IMT2022036
Harsh Vardhan Singh IMT2022101

Introduction

The objective of this project is to build a computer vision pipeline for face mask classification and segmentation. We implement traditional machine learning methods using handcrafted features, deep learning approaches using CNNs and U-Net, and compare their performances. The problem is relevant in health and safety monitoring scenarios, particularly in the context of public health crises like the COVID-19 pandemic.

Dataset

Two datasets were used:

- **Face Mask Classification Dataset:** Sourced from `https://github.com/chandrikadeb7/Face-Mask-Detection/tree/master/dataset`, containing images labeled as "with mask" and "without mask".
- **Masked Face Segmentation Dataset (MFSD):** Sourced from `https://github.com/sadjadrz/MFSD`, includes pixel-level annotations for face masks.

Methodology

Binary Classification using Handcrafted Features

1. **Data Preprocessing and Cleaning:** The dataset was organized into two categories: `with_mask` and `without_mask`. Some image files had problematic characters in their filenames, preventing them from loading properly. A renaming script was implemented to replace special characters (like `_¢`) with underscores, ensuring consistent loading and processing of all files.
2. **Feature Extraction Techniques:** Three types of handcrafted features were extracted from each image:
   - **HOG (Histogram of Oriented Gradients):** Captures the edge orientation and shape information.
   - **LBP (Local Binary Patterns):** Encodes local texture patterns into a histogram.
   - **Color Histogram (HSV):** Represents global color distribution in the image.

   These features were computed for resized grayscale/HLS images and concatenated to form a single feature vector per image.
3. **Data Splitting and Standardization:** The extracted features and labels were split into training (70%), validation (15%), and test sets (15%) using a two-step stratified `train_test_split`. Standardization was applied using `StandardScaler` to normalize feature values, improving convergence during classifier training.

4. **Classifier Training and Evaluation:** Three different machine learning classifiers were trained using the handcrafted features:

   - **SVM (Support Vector Machine):** Linear kernel, C=1.0
   - **MLPClassifier (Neural Network):** 2 hidden layers (128 and 64 neurons), trained for up to 500 iterations.
   - **XGBoost:** 100 trees, learning rate 0.1, depth 5.

   Models were evaluated on the validation set using the F1-score and classification report (precision, recall, accuracy).

5. **Final Testing:** Finally the F1-scores were calculated on the completely unseen testing data in order to compare with the performance of CNN used in the next section. The final F1-scores and full classification reports for the models are displayed in the *classification.ipynb* file, summarizing the performance across both classes.

Binary Classification using CNN

1. **Data Preparation and Renaming:** To avoid file loading issues, all image filenames in both `with_mask` and `without_mask` folders were renamed sequentially using a custom renaming function. The dataset was then loaded using TensorFlow's `image_dataset_from_directory`, with a 70%-30% training-validation split. The validation set was further divided equally into validation and test sets to ensure fair model evaluation.

2. **CNN Model Architecture:** A custom CNN model was defined using the Keras `Sequential` API. The architecture consisted of three convolutional blocks with increasing filters (32, 64, 128), each followed by Batch Normalization and MaxPooling. The output was flattened and passed through a dense layer, with a final sigmoid-activated node for binary classification. Binary cross-entropy loss was used, and the model was compiled with configurable optimizers. The CNN architecture chosen follows the standard protocol of increasing the number of filters as we progress to deeper layers and using 2*2 Max Pooling.

3. **Hyperparameter Tuning:** Extensive hyperparameter search was conducted over:

   - **Learning Rates:** `0.01`, `0.001`, `0.0001`
   - **Batch Sizes:** `16`, `32`
   - **Optimizers:** `Adam`, `RMSprop`
   - **Activation Functions:** `ReLU`, `Tanh`

   For each combination, a new model was trained for 15 epochs and validated using a separate validation set. TensorFlow's `AUTOTUNE` was used to optimize data pipeline performance with caching, shuffling, and prefetching.

4. **Model Evaluation:** After training, each model was evaluated on the unseen test set by computing the F1 score.. All results were stored in a list and later converted to a DataFrame for comparison and sorting.

Region Segmentation using Traditional Techniques

1. **Preprocessing and Input Preparation:** Images from the MSFD face crop dataset were first read and converted into grayscale format for efficient processing. A sample image was also visualized using Matplotlib to compare different segmentation outputs, including Otsu's thresholding and Canny edge detection.

2. **Otsu Thresholding and Morphological Operations:** Otsu's method was used to segment dark regions in the grayscale images by computing a global threshold automatically. To refine the resulting binary mask, morphological closing was applied which helps fill small holes and gaps in the segmented regions, creating a cleaner mask of the face mask area.

3. **Canny Edge Detection and Combination Strategy:** The Canny edge detector was used to capture the edges of mask regions. A combined mask was generated by taking the intersection of the Canny output and the closed Otsu mask, refining the mask boundaries further. This approach aims to blend the advantages of both intensity-based and edge-based segmentation.

4. **Batch Evaluation and Metric Computation:** All images in the dataset were processed to generate three predictions: Otsu-only, Canny-only, and the combined mask. Corresponding ground truth segmentation masks were loaded and binarized. For each method, segmentation masks were resized to match the ground truth, and two key metrics were computed:

   - **IoU (Intersection over Union)**
   - **Dice Coefficient**

   These metrics were calculated using custom functions and stored across all samples for averaging.

5. **Quantitative Results and Observations:** The evaluation results were averaged over all images:

   - **Otsu Thresholding:** Provided decent region segmentation with moderate overlap scores.
   - **Canny Detection:** Focused more on edges and underperformed in region coverage.
   - **Combined Mask:** Improved over Canny but did not outperform Otsu significantly.

   Overall we found that while traditional methods offered a basic segmentation capability, their effectiveness was limited by image variability, motivating the use of deep learning-based approaches like U-Net.

Mask Segmentation using U-Net

1. **Data Preparation and Preprocessing:** The face images and corresponding binary segmentation masks were loaded from the MSFD dataset. Each image and mask was resized to 256×256 resolution Pixel values of the images were normalized to the range [0, 1], while masks were binarized using a threshold of 127. The dataset was split into training (70%), validation (15%), and testing (15%) subsets using stratified sampling.

2. **U-Net Architecture:** A full U-Net model was implemented with a symmetric encoder-decoder structure. This structure has been exactly picked up from the original U-Net architecture that was proposed in the research paper with a small modification.
   After each convolution layer a Batch Normalization Layer was added to improve the stability of the model and increase training speed.

3. **Training Configuration:** The model was compiled with the Adam optimizer and binary cross-entropy loss. Custom metrics such as Dice coefficient and Intersection over Union (IoU) were defined to evaluate segmentation performance. The model was trained for 5 epochs using the preprocessed training and validation datasets, and monitored using both metrics.

4. **Evaluation and Metrics:** After training, the U-Net model was evaluated on the unseen test set.

Hyperparameters and Experiments

Due to hardware constraints, we were unable to perform exhaustive hyperparameter tuning on the CNN and U-Net architectures. Instead, we narrowed down the parameter space and limited our experiments to ensure they could run on our laptops within a reasonable time frame.

CNN Hyperparameter Tuning

To optimize the performance of the CNN model for binary classification, a grid search was conducted over the following hyperparameter space:

- **Learning Rates:** `0.01`, `0.001`, `0.0001`
- **Batch Sizes:** `16`, `32`
- **Optimizers:** `Adam`, `RMSprop`
- **Activation Functions:** `ReLU`, `Tanh`

Each combination was used to train a CNN model for 15 epochs on the training set, and evaluated using F1-score on a separate test set.

Best Performing Configuration

Among all trained models, the best configuration in terms of F1-score was:

- **Learning Rate:** 0.001
- **Batch Size:** 16
- **Optimizer:** Adam
- **Activation Function:** ReLU

This configuration achieved the following results on the test dataset:

- **Test F1-Score:** 0.9707

This demonstrates the importance of hyperparameter tuning, as the choice of optimizer, learning rate, and batch size significantly impacted the model's generalization performance.

U-Net Segmentation

Due to time and hardware constraints and the fact that the architecture of UNET is mostly defined in it's research paper, we only did the following experiments with UNET:

1. Tried a smaller version of Unet with less depth in the convolution blocks.
2. Tried the full version of Unet without Batch Normalization.
3. Tried the full version of Unet with Match Normalization.

The best results were obtained with the following configuration.

- Using BN after each convolution layer.
- Optimizer: Adam
- Loss Function: Binary Crossentropy
- Input Size: 256×256
- Batch Size: 16
- Epochs: 5

Results

Classification

The results obtained can be seen in Figure 1.

Model	Test Accuracy	Test F1-Score
Best CNN	0.970921	0.970684
Neural Network (MLP)	0.931707	0.923636
XGBoost	0.918699	0.906367
SVM	0.902439	0.891697

Figure 1: Classification Results

We can clearly see that Automatic Feature Learning/Extraction outperforms the models trained on Handcrafted Features.

Segmentation

The final results obtained for segmentation can be seen in Figure 2

We can clearly see that UNET performs significantly better than other traditional segmentation methods in terms of IOU and DICE scores.

Method	Average IoU	Average Dice
Otsu Thresholding	0.258274	0.360258
Canny Edge Detection	0.104900	0.182770
Otsu and Canny (Combined)	0.060586	0.107802
U-Net	0.614172	0.760439

Figure 2: Segmentation Results

Observations and Analysis

- CNN outperformed traditional classifiers(With a *F1 of 0.97 v/s 0.92*) in classification tasks made using Handcrafted Features.
- This implies that Automatic Feature Learning has a clear advantage over Handcrafted Features.
- U-Net outperformed traditional segmentation methods by a large margin.(*IOU of 0.76 v/s 0.36*)
- This is because traditional methods are sensitive to image conditions and do not generalize well across diverse data.
- One of the challenges faced during performing the classification tasks was the names of certain Images.Many images had names with symbols not consistent with utf-8 format, making reading them difficult.In order to solve this, we renamed all the images sequentially and then trained out models.
- Another challenge was the lack of computation power at our disposal that prevented us from running more experiments.

How to Run the Code

1. Clone the repository and install the required packages using `pip install -r requirements.txt`.
2. Run `classification.ipynb` for both handcrafted and CNN-based classification.
3. Run `segmentation.ipynb` for traditional and U-Net-based segmentation.
4. All outputs and performance metrics are displayed in the respective notebooks.

Directory Structure

The project follows a simple and organized directory structure as shown below.

```
classification_dataset/     # Contains images for classification (with_mask, without_mask)
MSFD/                       # Contains MSFD segmentation images and masks

classification.ipynb       # Jupyter notebook for classification (handcrafted + CNN)
segmentation.ipynb         # Jupyter notebook for segmentation (traditional + U-Net)
Readme.pdf                 # PDF version of the Readme contained all the required sections
requirements.txt           # All the packages needed to run the code
```

Each component is modularly separated for ease of experimentation and reproducibility.