Visual Recognition: Coin Counting and Image Stitching

Aaryan Dev, Shashank Devarmani, Soham Pawar

March 25, 2025

Abstract

This assignment is divided into four major tasks. These tasks range in increasing complexity, starting from binary classification of input image(s) to mask segmentation using U-net. For each of the tasks, we try to describe the method used, and the results obtained by using different model architectures for the tasks.

1 Task 1

Binary Classification Using Handcrafted Features and ML Classifiers. This task mainly aims at the binary classification of an input image into whether the person in the image is wearing a mask. The dataset used for this task is from this GitHub repo. The dataset contains two types of images, stored in two folders with the respective names; images of people with mask and without mask.

1.1 Extracting handcrafted features

For this task, we consider two types of features; Histogram of Oriented Gradients (HOG) and Histogram of Canny Edge detection features.

- The HOG features for the input images are extracted using the scikit-learn's in-built function. This captures local edge patterns by computing gradient orientations in small regions of an image. For the task, normalization was done considering 2x2 cells together. Each cell is a block of 8x8 pixels. For calculating the gradients, each gradient is divided into 9 bins between 0 and 180 degrees. This method captures edge and texture details which is used for classification of the imaged using different models.
- The Canny edge features for the input images are extracted using OpenCV's built-in Canny edge detection function. This method identifies edges by detecting intensity changes and suppressing weaker ones to highlight only the strongest edges. For this task, the edge detection thresholds were set to 50 and 150. Once the edges are detected, a histogram of pixel intensities is generated with 256 bins, covering values from 0 to 255. The histogram is then normalized to represent the distribution of edge intensities. These features help capture the prominence of edges in the image and are used for classification with different models.

1.2 Training and Evaluating models

- For both the hand-crafted features, we train a SVM with a linear classifier. This is easily done using the scikit-learn SVC (State Vector Classifier) function available under the SVM package. The inputs for each of the features is generated by iterating over the images of each of the class (mask and non-mask) and storing the histogram of the features in a array. This array is converted into the compatible numpy array format. This is then split into train and test arrays with an 80-20% break-up.
- Like the SVM Classifier, a Multi-Layer Perceptron (MLP) model with two hidden layers is used. These hidden layers have 128 and 64 number of neurons. This model trains for a maximum of 500 epochs or lesser if convergence is achieved earlier. The same train-test split is used for training and evaluating the model.

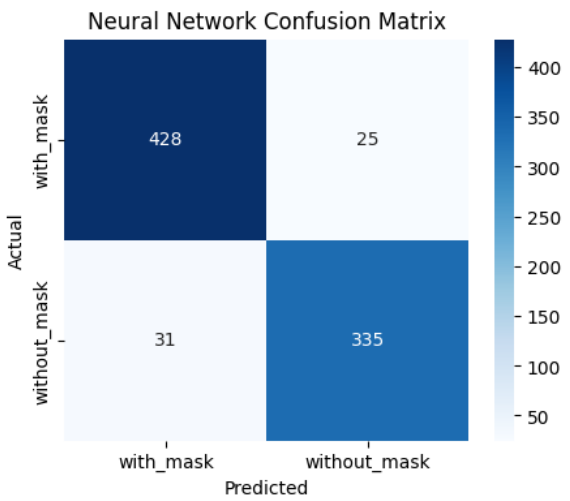

Figure 1: Confusion matrix representing the results of MLP classifier on HoG features

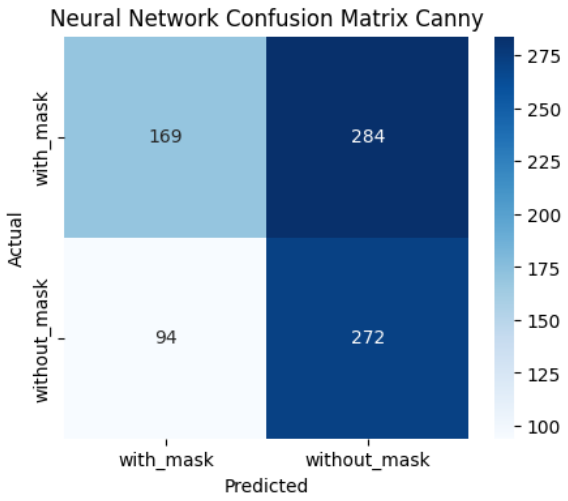

Figure 2: Confusion matrix representing the results of MLP classifier on Canny edge features

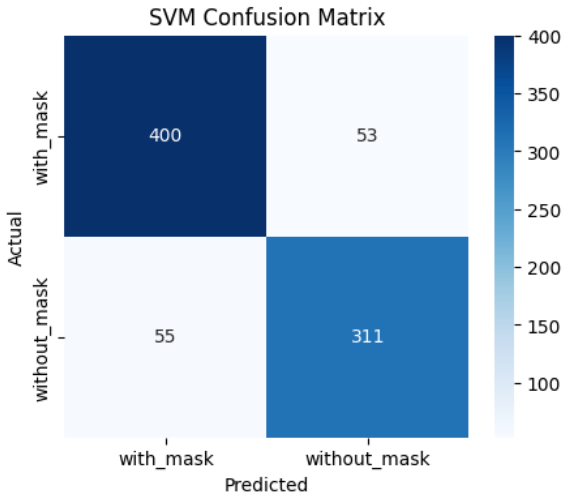

Figure 3: Confusion matrix representing the results of SVM classifier on HoG features

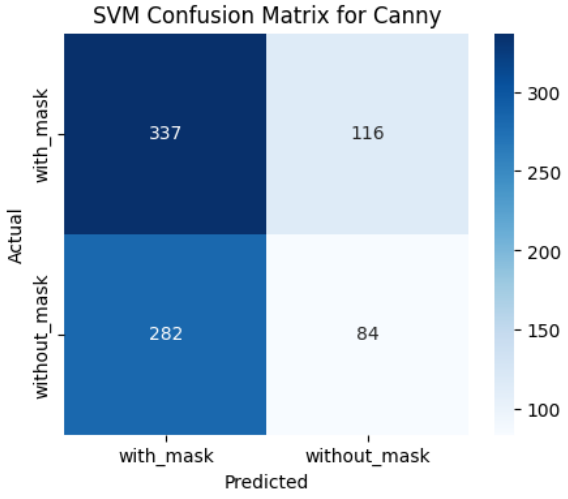

Figure 4: Confusion matrix representing the results of SVM classifier on Canny edge features

1.3 Results and Comparison

It can be seen from the confusion matrices for the two types of features and two types of classifiers that the HoG features are better at predicting the class of the input image than the Canny edge features. Between the type of classifier, the MLP (NN-based) classifier is more accurate in predicting the classes than the SVM classifier. From all these, the best accuracy is from the MLP Classifier for HoG features with accuracy of 93.2% and recall of 94.5% for 'with mask' class and 91.5% for 'without mask' class.

2 Task 2

Binary Classification Using CNN. This task also aims at the binary classification of the input image(s) into the classes: with mask or without mask. This task uses the same dataset used in Task 1. The only difference in this task is that of the approach used. In Task 1, handcrafted features were to be used which was given as input to ML classifiers. However, in this task, the features are to be learned using the convolution layers of a Convolution Neural Network (CNN) rather than using hand-crafted ones. Using CNNs provides higher flexibility to feature learning and is more data-driven although more computationally intensive.

2.1 Designing the model

Unlike the ML classifiers in Task 1, the features are learnt my the NN model using the convolution layers. In our model, we firs have some convolution layers and then the final convolution layer's outputs are flattened and then passed onto a FFN (Feed-Forward Network). The ouput from the final layer of the FFN is the final output which does the classification.

- The whole NN model can be divided into two main parts of the model. The first part of the model can be seen as a combination of convolutional layers using different-sized kernels. After each convolution layers, we scale-down the feature images using Average Pooling. This is then passed on to the activation function which forms a form of 'feature image' for the next convolution layer-sampling down operation set. The model takes as input images in the RGB/BGR format which means 3 channels. In our model, we use two convolution layers, both with a 3x3 kernel. The first layers gives the output as 32 channels and the second layer gives the output as 64 channels. For both the convolution layers, the ReLU activation function is used after the Max Pooling kernel to sample-down the feature images after the convolution operation.
- After the final convolution layer-pooling set, the output of the final activation function is first flattened. This is so that it can be given as input to the sequential FFN for the final classification. After flattening the output from the final convolution operation set, a fully connected layer with 64 neurons is used. The activation function used for this Dense layer is again the ReLU function. After this layer, a Dropout is used with the probability of 0.5. This dropout is applied to ensure that there is no over-fitting of the model. After this fully-connected layer, the final layer contains only 1 neuron as the output. This is because of the problem being a binary classification problem. Thus, the final activation function used is the Sigmoid activation fucntion.

2.2 Training the model

For the training of the model, we experiment with two optimizers: Adam and SGD optimizers. For the Adam optimizer, the learning rate is 0.001. For the SGD algorithm, the learning rate is 0.001 and the momentum is 0.9. We also change the activation functions between the Sigmoid and TanH functions. The dimension of the input image is 128x128 with 3 channels (RGB/BGR). The batch size used is 32 and 64. We also vary the learning rate of the optimizers. The model is trained for 20-30 epochs with early stopping on validation accuracy. The results can be summarized in Table 1.

2.3 Comparison with the ML Classifiers

For evaluating the model's predictions, we use the scikit-learn's precision_recall_curve function. This returns the precisions and recalls values to calculate the F1 scores. The threshold values are compared

Table 1: Summarization of the results of model training with varying hyper-parameters

Activation	Optimizer	Batch Size	LR	Momentum	Train Acc.	Val. Acc.
Sigmoid	Adam	32	0.001	-	99.7	98.17
Sigmoid	SGD	32	0.001	0.9	95.95	95.72
Sigmoid	SGD	32	0.005	0.9	94.52	96.33
Sigmoid	Adam	32	0.005	-	98.63	95.74
Sigmoid	Adam	64	0.001	-	99.7	97.43
TanH	Adam	32	0.001	-	98.94	98.19

based on the calculated F1 scores to determine the best model. It can be clearly seen from the table in Task 2 and from the confusion matrices from Task 1 that the NN model with the CNN layers and then the FFN is better at classifying the input images.

3 Task 3

3.1 Implementation of a Region-Based Segmentation Method

Region Segmentation Using Traditional Techniques To segment mask regions for faces labeled as "with mask," we employ **HSV-based thresholding**, a widely used region-based segmentation approach. The method converts the input image from **RGB to HSV color space**, allowing more effective segmentation by focusing on hue, saturation, and value levels rather than raw RGB intensity.

We systematically **search for optimal threshold values** by iterating over a predefined range of **hue (H), saturation (S), and value (V)**. This exhaustive search ensures that the segmentation performance is maximized by minimizing the **Mean Squared Error (MSE)** between the generated mask and the ground truth segmentation.

The workflow for the segmentation method is as follows:

1. Convert the input image to **HSV color space**.
2. Define **lower and upper bounds** for HSV values that represent the masked region.
3. Use `cv2.inRange()` to generate a **binary mask**, where pixels within the threshold range are marked as foreground (mask region) and others as background.
4. Compare the segmented mask with the ground truth using **MSE** to evaluate segmentation performance.

3.2 Visualization and Evaluation of Segmentation Results

The segmentation results are visualized by overlaying the generated binary mask onto the original image. This allows a qualitative assessment of whether the segmentation method correctly identifies the masked regions.

For **quantitative evaluation**, we compute the **Mean Squared Error (MSE)** between the predicted mask and the ground truth segmentation mask. MSE is calculated using the following equation:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(M_i - G_i)^2 \tag{1}$$

where:

- M_i is the pixel value in the predicted mask,
- G_i is the corresponding pixel value in the ground truth mask,
- N is the total number of pixels.

The **optimal threshold values** are determined based on the lowest observed MSE, ensuring the best segmentation accuracy.

The final output includes:

- **Best HSV threshold values** identified through optimization.
- **MSE score**, indicating segmentation accuracy.
- **Visualizations** of segmented images for qualitative assessment.

By employing HSV thresholding, this method provides an efficient and **traditional** alternative to deep-learning-based segmentation, particularly in scenarios where computational resources are limited.

4 Task 4

Mask Segmentation Using U-Net. This task requires building a U-Net model for precise segmentation of mask regions in the images. To better do the task, we have created individual python file for separate tasks.

4.1 The U-Net Model

The U-Net model is designed for precise segmentation of mask regions in images. It is a fully convolutional neural network that follows an encoder-decoder structure, making it highly effective for pixel-wise classification tasks. The primary objective of this model is to accurately delineate mask regions in input images by learning spatial features and reconstructing detailed segmentation maps.

- **Encoder (Contracting Path):** The encoder extracts hierarchical features using a series of convolutional layers followed by max-pooling operations. Each block in the encoder consists of two convolutional layers with ReLU activation, followed by a downsampling step that reduces spatial dimensions while increasing feature depth. This step captures high-level representations necessary for segmentation.
- **Bottleneck Layer:** At the bottom of the U-Net structure, the bottleneck layer acts as a bridge between the encoder and decoder. It consists of convolutional layers with high feature depth but smaller spatial dimensions, allowing the network to learn complex, abstract representations of the input.
- **Decoder (Expansive Path):** The decoder reconstructs the segmented mask by progressively upsampling the feature maps. Each block in the decoder consists of an upsampling operation followed by two convolutional layers with ReLU activation. Skip connections from the corresponding encoder layers are concatenated to retain fine spatial details lost during downsampling. This enables precise localization of mask regions.
- **Output Layer:** The final layer applies a 1×1 convolution followed by a sigmoid (for binary segmentation) or softmax activation (for multi-class segmentation). This converts the feature maps into a segmentation mask where each pixel represents a class probability.

4.2 Training and Optimization

- The model is trained using a dataset consisting of images and their corresponding ground truth segmentation masks. The input images are preprocessed through resizing to a standard size of 128 x 128. Since the dataset to be used is not directly given as input to the model for the training, a custom dataset is created as part of the dataset.py python script.
- For the training process, we use the BCE (Binary Cross Entropy) loss function. This is available as the BCEWithLogitsLoss available as an in-built function in pytorch. As for the optimizer for the training process, we choose the Adam optimizer. We train the model for 40 epochs with the mentioned loss and optimizer functions.
- All the hyperparameters for the model training and definitions are defined in the config.py python script. This helps in simplified access and updates of the parameters in a single place. This file is accessed in multiple files related to model definition, training and inference.

4.3 Inference

During the inference of the model, the trained model is given as input the image path. The image is read using the input image path. All the image reading and transformations on the image are done using opencv's in-built functions and with simplified array interaction using numpy. For each of the input images, two scoring metrics are used to evaluate the model's performance. These are the IoU and Dice loss functions. These are also plotted for better visualization. All of the functions required for the calculation are clearly defined in the prediction.py python script.