EE655: Computer Vision and Deep Learning

ASSIGNMENT - I

Name: Vaibhav Itauriya

Roll Number: 231115

Branch: ME

Date of submission: 15/02/2025

GitHub Repository of this Course:

Contents

1 Smile Detection Using Key Points of the Mouth

Detecting whether a person is smiling can be achieved using four key points of the mouth:

- C_L: Left corner of the mouth
- C_R: Right corner of the mouth
- T: Top point of the mouth
- B: Bottom point of the mouth

Based on these points, we define three features to determine if a person is smiling.

The following features are used to detect a smile:

1.1 Mouth Width and Height

The width and height of the mouth are computed as:

$$\text{Mouth Width} = ||C_R - C_L|| \tag{1}$$

$$\text{Mouth Height} = ||T - B|| \tag{2}$$

A smile is likely if the width is significantly greater than the height:

$$\text{Smile Indicator} = 1(\text{Mouth Width} > \text{Mouth Height}) \tag{3}$$

1.2 Mean Corner Elevation

The midpoint between the top and bottom lips is computed as:

$$\text{Mid Lip} = \frac{T + B}{2} \tag{4}$$

The elevation of the mouth corners is given by:

$$\text{Left Corner Elevation} = C_L^y - \text{Mid Lip}^y \tag{5}$$

$$\text{Right Corner Elevation} = C_R^y - \text{Mid Lip}^y \tag{6}$$

A smile is detected if the average elevation is positive:

$$\text{Smile Indicator} = 1\left(\frac{\text{Left Corner Elevation} + \text{Right Corner Elevation}}{2} > 0\right) \tag{7}$$

1.3 Width-to-Height Ratio

Another useful feature is the width-to-height ratio:

$$\text{Ratio} = \frac{\text{Mouth Width}}{\text{Mouth Height}} \tag{8}$$

A higher ratio indicates a smile.

1.4 Implementation and Code

Below is the Python implementation of these features:

```
import numpy as np

def compute_smile_features(C_L, C_R, T, B):
    C_L, C_R, T, B = map(np.array, [C_L, C_R, T, B])
    mouth_width = np.linalg.norm(C_R - C_L)
    mouth_height = np.linalg.norm(T - B)
    is_smiling_width = int(mouth_width > mouth_height)
    mid_lip = (T + B) / 2.0
    left_corner_elev = C_L[1] - mid_lip[1]
    right_corner_elev = C_R[1] - mid_lip[1]
    mean_corner_elev = 0.5 * (left_corner_elev + right_corner_elev)
    is_smiling_elev = int(mean_corner_elev > 0)
    width_height_ratio = mouth_width / mouth_height
    return is_smiling_width, is_smiling_elev, width_height_ratio
```

Here is the complete code for the given question:

Code Link: Click Here

1.5 Conclusion

Using basic geometric calculations, these three features provide a simple yet effective way to determine if a person is smiling. Further enhancements can include machine learning techniques to improve accuracy.

2 Modified LeNet Architecture for MNIST Classification

In this question, the following modifications were incorporated:

- Inclusion of a softmax layer at the end.
- Use of $x \cdot \sigma(x)$ as the activation function instead of ReLU.
- Replacement of average pooling with max pooling.
- Use of only 3×3 filters in convolutional layers.

2.1 Dataset and Preprocessing

The MNIST dataset, containing 60,000 training images and 10,000 test images of handwritten digits (0-9), was used. Standard normalisation techniques were applied to scale pixel values to the range $[0, 1]$. The dataset was preprocessed as follows:

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from TensorFlow.keras.datasets import most
import numpy as np
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

2.2 Model Architecture

The modified LeNet architecture consists of:

- Two convolutional layers with 3×3 filters.
- Max pooling layers for downsampling.
- Fully connected layers leading to a softmax output.

The custom activation function used was:

$$f(x) = x \cdot \sigma(x) \tag{9}$$

where $\sigma(x)$ is the sigmoid function.

2.3 Implementation

The model was implemented in Tensorflow as follows:

```python
def custom_activation(x):
return x * tf.math.sigmoid(x)

model = models.Sequential([
layers.Conv2D(6, (3, 3), activation=custom_activation, input_shape=(28,
    28, 1)),
layers.MaxPooling2D((2, 2)),
layers.Conv2D(16, (3, 3), activation=custom_activation),
layers.MaxPooling2D((2, 2)),
layers.Flatten(),
layers.Dense(120, activation=custom_activation),
layers.Dense(84, activation=custom_activation),
layers.Dense(10, activation='softmax')
])
```

2.4 Training

The model was trained using categorical cross-entropy loss and the Adam optimiser:

```
model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(
    x_test, y_test))
```

2.5 Code

Here is the complete code for this question:

Code Link: Click Here

2.6 Results and Conclusion

The modified LeNet achieved an accuracy of 98.92% on the MNIST test set. Using $x \cdot \sigma(x)$ as an activation function resulted in smooth gradients, and max pooling improved feature selection.

3 Modified Histogram of Oriented Gradients (HoG) with Roberts Cross Edge Detector

For this, a modified Histogram of Oriented Gradients (HoG) feature extraction algorithm is implemented. The main modifications include:

- Using the Roberts cross-edge detector for computing image gradients.
- Extracting HoG features from images in the Cat and Dog dataset.
- Training a Random Forest classifier on the extracted features.

3.1 Dataset and Preprocessing

The dataset consists of images of cats and dogs. The images were preprocessed as follows:

- Resized to a standard dimension for uniformity.
- Converted to grayscale for simplicity.
- Normalized for numerical stability.

3.2 Feature Extraction with HoG

HoG is a widely used feature descriptor that captures object shape and texture. The key steps involved in computing HoG features using the Roberts cross-edge detector are:

1. Compute gradients using the Roberts cross operator:

$$G_x = I(x,y) - I(x+1,y+1), \quad G_y = I(x,y+1) - I(x+1,y) \tag{10}$$

2. Compute gradient magnitude and orientation:

$$M = \sqrt{G_x^2 + G_y^2}, \quad \theta = \tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{11}$$

3. Divide the image into cells and compute a histogram of orientations.

4. Normalize histograms across blocks for contrast invariance.

3.3 Training the Classifier

A Random Forest classifier was trained on the extracted HoG features. The classifier was chosen due to its robustness and ability to handle high-dimensional feature spaces.

3.4 Evaluation and Results

The classifier was evaluated using accuracy metrics, and it successfully distinguished between cat and dog images based on HoG features. We have achieved an accuracy of 70.79% only.

3.5 Code

The full implementation of this modified HoG feature extraction and classification can be found in the following GitHub repository:

Code Link: Click Here

4 Object Counting in Binary Images using BFS

4.1 Algorithm Explanation

To count the number of distinct objects in a binary image, we utilise a Breadth-First Search (BFS) approach. The image is represented as a 2D matrix of pixels, where foreground objects are marked in white (255) and the background is black (0). The algorithm follows these steps:

1. Read the binary image and ensure it is properly thresholded to distinguish objects from the background.

2. Initialize a visited matrix of the same size as the image to keep track of explored pixels.

3. Define the 8-connected neighbourhood, allowing horizontal, vertical, and diagonal movement.

4. Iterate through each pixel in the image:

    - A new object is detected if an unvisited foreground pixel (255) is found.
    - Initiate a BFS from this pixel, marking all connected pixels as visited.
    - Increment the object count.

5. Continue until all pixels have been processed.
6. Return the total object count.

4.2 Challenges Faced

- **Noise in the Image:** Inconsistent thresholding or artefacts may lead to incorrect object detection.
- **Handling Large Images:** BFS requires additional memory to maintain the queue, which may become inefficient for huge images.
- **Connected Component Complexity:** Properly managing diagonal connections to avoid double-counting objects.
- **Performance Optimization:** Ensuring the algorithm runs efficiently on different image resolutions.

4.3 Code Implementation

The full implementation of the BFS-based object counting algorithm can be found at the following link:

Code Link: Click Here