Deep Vision Crowd Monitor: AI for Density Estimation and Overcrowding Detection

Intern Name: Ishu.

Table Of Content

1. Project Overview and Dataset

1.1. Project Description

The objective of the Deep Vision Crowd Monitor: AI for Density Estimation and Overcrowding Detection project is to develop a real-time deep learning-based system capable of accurately estimating crowd density and detecting overcrowded zones using surveillance video feeds. This system aims to enhance public safety, support emergency response, and optimize crowd flow management in high-footfall areas such as transit hubs, public events, and smart city infrastructures. By leveraging convolutional neural networks (such as CSRNet or MCNN) and advanced crowd estimation algorithms, the project seeks to provide timely insights and automated alerts to authorities, enabling proactive interventions and efficient crowd control. The overall workflow includes video input, frame extraction, preprocessing, crowd estimation, counting and detection, and real-time alert generation.

1.2. Dataset Used

This project utilizes a standard academic dataset for crowd counting, inferred to be the **ShanghaiTech Dataset** (specifically, Part A or Part B, depending on the environment). This dataset consists of a collection of crowd images and corresponding ground truth data. Each image captures a specific crowd scene. The ground truth is provided as a set of coordinate points (x, y) for every visible person in the image, typically stored in a .mat (MATLAB format) file. These point coordinates are crucial for the primary task, which is generating density maps for training the deep learning model. The crowd counts per image can range from very sparse to extremely dense, allowing the model to be trained on a wide variety of crowd scenarios.

2. Environment Setup Theory

The foundation of this project is built upon a standard Python environment tailored for deep learning and computer vision tasks. The environment setup requires the installation of several key libraries, which can be managed efficiently using a requirements.txt file.

• *Deep Learning Framework*: PyTorch and torchvision are necessary for developing, training, and running the deep convolutional neural network (CNN) model (e.g., CSRNet).

• *Computer Vision*: opencv-python is used for reading, manipulating, and writing image and video data.

• *Scientific Computing*: numpy and scipy (specifically for loading .mat files and applying Gaussian filters) are core libraries for data manipulation and numerical operations.

• *Visualization and Utilities*: matplotlib and Pillow are used for data exploration and visual confirmation, while pandas and tqdm are used for data management and progress tracking.

• *Optional Performance Boost*: If the system includes an NVIDIA GPU, installing the relevant CUDA toolkit and drivers is essential to accelerate the intensive model training and inference processes.

3. Data Exploration Stage

Data exploration focuses on understanding the raw input data (image and ground truth points) and transforming the discrete point data into a continuous density map, which serves as the label for training the CNN.

3.1. Ground Truth Analysis

The primary step is loading the raw image file (e.g., JPEG or PNG) and the corresponding ground truth .mat file. The .mat file contains the pixel coordinates $P = \{(x_i, y_i)\}_{\{i=1\}}^{\{N\}}$

for all N people in the image. The simple crowd count is the number of points, N.

3.2. Density Map Generation

A deep learning model cannot be trained to regress discrete points; it must regress a continuous map. The core task in this stage is generating a density map D where the integral over any region gives the count of people in that region. This is typically achieved using a **Gaussian Kernel.**

1. *Initial Delta Map*: A sparse map M is created, the same size as the input image, where a value of 1 is placed at every ground truth coordinate (x_i , y_i), and 0 everywhere else.

2. *Gaussian Filtering*: The sparse map M is convolved with a Gaussian kernel $G\sigma$ with a standard deviation σ:

$$D = M * G\sigma$$

The standard deviation σ is a hyperparameter that controls how much a single point is spread across its local area. A larger σ results in smoother, more diffused density blobs.

The resulting density map D is a floating-point matrix where the sum of all elements should approximate the total crowd count N. The exploration stage concludes by visualizing this density map, often as a heatmap overlaying the original image, to visually confirm that density peaks align with the crowd locations.

4. Data Preprocessing Stage

The preprocessing stage prepares the raw density maps and images for consumption by a deep learning model, ensuring uniformity and stability across the entire dataset.

4.1. Initial Loading and Format Conversion

This stage begins with loading both the image and the corresponding ground truth (GT) density map file (from .npy or .mat format). For the image data, the industry standard cv2.imread function is used, which loads the image in BGR (Blue-Green-Red) channel order by default. This must immediately be converted to the RGB format using cv2.COLOR_BGR2RGB. The pixel data is then converted to a floating-point representation (float32) and scaled to the 0–1 range by dividing all values by 255.

4.2. ImageNet Normalization

Following the initial scaling, the image is prepared for input into the neural network by applying a standard normalization technique commonly used with models pre-trained on ImageNet. This involves subtracting the channel-wise mean (μ) and dividing by the standard deviation (σ):

$$I_{normalized} = (I_{raw} - \mu)/ \sigma$$

The standard values used are:

- *Mean*: [0.485, 0.456, 0.406]
- *Standard Deviation*: [0.229, 0.224, 0.225]

This process, typically handled efficiently by transforms. Normalize in PyTorch, centralizes the data distribution, which aids in stable model convergence.

4.3. Density Map Downsampling and Scaling

The final ground truth density map must match the reduced feature size of the model's output layer (often 1/8th of the input image size).

1. Downsampling: The GT density map (which is already in float32) is downsampled by a factor of 8 × 8. This spatial reduction is achieved using cv2.resize with the INTER_AREA interpolation

method, which is preferred for image decimation as it prevents aliasing artifacts and produces clearer results.

2. Count Preservation Scaling: Crucially, when the map is downsampled by a factor of 8 in both height and width, the total area is reduced by 8 × 8 = 64. To ensure that the fundamental truth– the total crowd count –is preserved, every value in the downsampled density map must be multiplied by a factor of 64: $D_{final} = D_{downsampled} \times 64$.

4.4. PyTorch Data Structure and Tensor Conversion

The final processed image and density map must be converted into PyTorch Tensor objects to be consumed by the deep learning model.

• *Image Tensor*: The NumPy image array (H × W × C) is converted to a PyTorch tensor and its dimensions are permuted to the standard channel-first format (C × H × W).

• *Density Tensor*: The NumPy density array (H × W) is converted to a PyTorch tensor and a channel dimension is added using unsqueeze(0), resulting in a (1 × H × W) shape.

• *Dataset and DataLoader*: The tensors are organized into a custom PyTorch Dataset class, which properly implements __len__ and __getitem__. The DataLoader is then built to handle batching, shuffling, and multi-process data loading (num_workers), with pin_memory=True recommended for efficient GPU transfer.

4.5. Output Data Storage

Upon successful completion of the preprocessing steps, the final, ready-to-use data is systematically saved to a designated output directory, separating it from the raw dataset. This output is stored in formats highly optimized for the PyTorch deep learning framework.

• *Tensors (.pt files):* The normalized image data (input) and the downsampled, scaled density map data (label) are stored as PyTorch tensor files (.pt extension). Storing the data directly as tensors eliminates the need for repeated loading and transformation during the training phase, significantly speeding up data loading within the training loop.

• *NumPy Arrays (.npy files)*: As an intermediate or verification step, the final density maps are also often saved as NumPy arrays (.npy extension). This format is useful for quick visual inspection, debugging, and verification of the crowd count integrity before model consumption.

All these files are organized into separate folders (e.g., tensors/images and tensors/density_maps) within the master output directory, along with a manifest file that maps each image ID to its corresponding tensor and density file path.