VIP-IPA: Pedestrian Detection Accident Avoidance

Donny Weintz, Aditya Mallepalli, Jianing Wang, Preetham Yerragudi, Aakarsh Rai

Advisors: Edward J. Delp, Carla Zoltowski

December 6, 2024

TABLE OF CONTENTS

1 ABSTRACT

Pedestrian detection is a critical component in road crash avoidance systems, especially with the increasing interest in autonomous vehicles and advanced driver assistance systems. Accurately detecting and localizing pedestrians in real-time is essential for ensuring the safety of both pedestrians and drivers.

This project aimed to develop an effective pedestrian detection system using images taken from a moving vehicle as input. The system should include a mechanism to guide the direction of steering for autonomous vehicles in near-crash situations so that the car can avoid as many pedestrians as possible.

To train and test our model, we utilized a dataset that includes images taken from the vehicle dashboard camera in various cities throughout Europe. The data set included $3,000+$ images, of which 83% was used for training and 17% for testing. We used histograms of oriented gradients to extract features from labeled pedestrians within the dataset and randomly sampled areas that are known to not contain pedestrians. These features were fed into a support vector machine model to differentiate pedestrians from non-pedestrian objects with high accuracy. Then a sliding-window approach was applied to scan images at varying scales and locations to detect pedestrians of any size and position.

We implemented optimizations using a combination of hard negative mining, color-filtering, threading, and non-maximum suppression to reduce false positive detections and decrease runtime. Taking the final output, we derived an algorithm that produced the best possible direction for the vehicle to move in that would result in the least number of pedestrian collisions.

2 DATASET

To find a suitable dataset for our project that could accurately train a model, we needed one that met the following criteria:

- Images were taken from the dashcam of a moving vehicle
- A sufficiently large number of samples $(3,000+)$
- Annotations containing the location and size of each pedestrian
- Split into training and validation sets

2.1 WIDER Dataset

The first compatible dataset we found was used in the WIDER Face and Person Challenge held in 2019 [1]. This dataset included images taken from both surveillance cameras and the dashcam of a vehicle, however, we are only interested in the latter. $88,260$ dashcam images were provided, $2,519$ of which were to be used for the validation of the model.

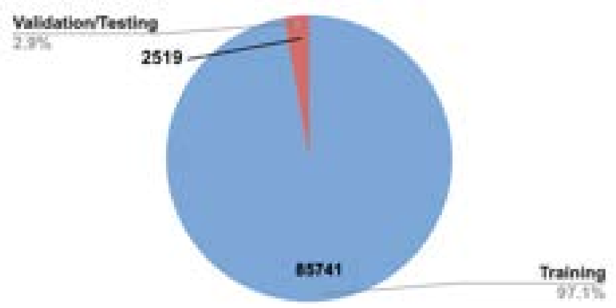

Figure 2.1: WIDER dataset training vs. validation split

An annotated sample from the dataset is shown in Figure 2.2.

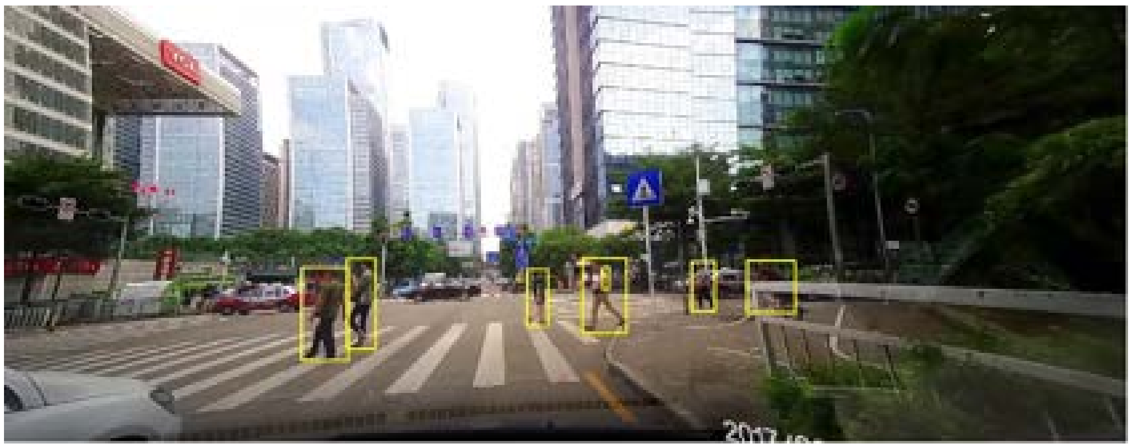
Figure 2.2: Annotated ground truths from WIDER dataset

Although this dataset met the basic requirements for what we were looking for, we found working with it difficult due to various factors. The images were taken with a wide-angle camera, making each pedestrian appear warped and smaller than they normally would, thus harder to detect. Different lighting conditions due to rain or night-time also contributed to making our model less consistent than we would have liked. For now, accounting for these lighting conditions was beyond the scope of our project, and we instead looked for a more consistent and easier-to-work-with dataset.

2.2 Cityscapes Dataset

We found the Cityscapes dataset, which is a popular dataset used commonly for the semantic understanding of urban street scenes, including annotations for many different types of objects on the road, including pedestrians [2]. The dataset provided $3,475$ images, 500 of which were to be used for validation.

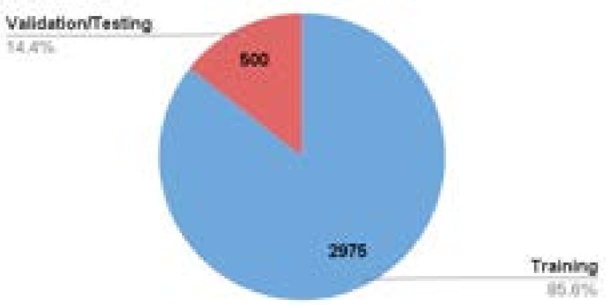

Figure 2.3: Cityscapes dataset training vs. validation split

An annotated sample from the dataset is shown in Figure 2.4.

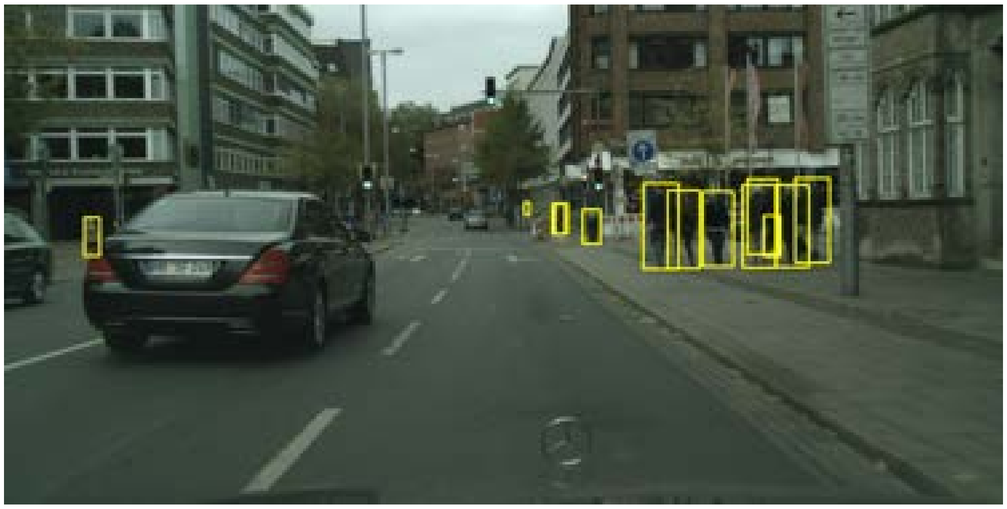

Figure 2.4: Annotated ground truths from Cityscapes dataset

We found this dataset far more suitable for the goal of our project, as the images were taken without a wide-angle lens, making each pedestrian easier to detect. Additionally, the coloring/lighting conditions were consistent across every image, and each one generally contained less noise than the WIDER dataset.

3 HISTOGRAMS OF ORIENTED GRADIENTS

Before creating a pedestrian detector, it is important to preprocess and extract the most relevant information from the image. When detecting pedestrians, the most important features in the image are the edges and shapes. Features such as color and texture are not as important.

Histograms of oriented gradients (HOG) is a feature descriptor commonly used in object detection. First introduced by Dalal and Triggs in 2005 [3], HOG captures local object appearance and shape by analyzing the distribution of gradient orientations in localized image regions. The feature descriptor generated through HOG can then be used in conjunction with a Support Vector Machine (SVM) to develop a pedestrian detector. The steps we used to generate HOG features are described below.

3.1 Preprocessing

The first step in computing a HOG feature descriptor is to preprocess the image. The image must have a fixed aspect ratio. In our implementation, an aspect ratio of $1:2$ is used in all HOG computations. The image must also contain a clear pedestrian in order for the extracted features to be usable. The dataset used in our implementation had predefined bounding boxes enclosing each pedestrian. We extracted the pedestrian from the bounding box and resized it to a size of 64×128 pixels. This ensured that each pedestrian could be described by a consistent-sized HOG feature vector, which is critical for the application with SVM.

3.2 Calculating Gradients

Once an image has been preprocessed, the horizontal and vertical gradients must be computed. The gradient operation is used to find the edges in an image by analyzing the change in intensity of adjacent pixels. The computation can be performed through many different gradient window operations. In our implementation, we used the following windows:

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

Figure 3.1: Illustration of horizontal and vertical kernels. The kernel for the gradient in the x direction is shown on the left and the kernel for gradient in the y direction is shown on the right.

Once the horizontal and vertical gradients are computed, we must determine their magnitude and direction. The formula for the magnitude of the gradient is shown in Equation 3.1, where g_x is the gradient in the x direction, g_y is the gradient in the y direction, and g is the magnitude of the gradient.

$$g = \sqrt{g_x^2 + g_y^2} \tag{3.1}$$

The formula for the direction of the gradient is shown in Equation 3.2, where θ is the direction of the gradient. In the histograms of orientation gradients, the direction is represented as unsigned, which means that all direction values should be interpreted as being between $0°$ and $180°$.

$$\theta = arctan(\frac{g_y}{g_x}) \tag{3.2}$$

3.3 Creating Histograms

After the gradients in the image are computed, we break the image into cells. In each cell, we create a histogram of the oriented gradients. In our HOG implementation, we used a cell size 8×8 pixels, where each image was sized to 64×128 pixels. This allowed us to have 128 total cells in each image.

To create histograms, we initialize a histogram for each cell that contains 9 bins. The bin values range from $0°$ to $160°$, and each bin value is an increment of $20°$. For each cell's histogram, the gradient magnitude values are placed according to their corresponding directions. If the direction of the gradient is between two bin values, it will split into each bin depending on the distance to each bin.

For example, if the gradient magnitude was 100 in a direction of $65°$, 25 would be added to the bin at $60°$ and 75 would be added to the bin at $80°$. This is because $65°$ falls between the bin values of $60°$ and $80°$. Since $65°$ is closer to the bin at $60°$, more of the magnitude is distributed to that bin. The formulas used to calculate the contributions of the bin are detailed in Equations 3.3, 3.4, and 3.5 where $lower\ contribution$ is the magnitude portion that falls in the lesser bin and $upper\ contribution$ is the magnitude portion that falls in the larger bin.

$$remainder = \theta \;\%\; 20 \tag{3.3}$$

$$lower\ contribution = g \cdot (remainder/20) \tag{3.4}$$

$$upper\ contribution = g \cdot ((20 - remainder)/20) \tag{3.5}$$

The histograms are computed for each cell, and the 9 values in each histogram can be used to describe the features of that cell. These values are the strength of the gradient in the 9 directions specified in the histogram. An example histogram using real data from our analysis is shown below. The image in Figure 3.2 is a pedestrian in which we computed HOG features. The image in Figure 3.3 is one of the 8×8 pixel cells used to compute HOG. The histogram of oriented gradients for this cell is illustrated in figure 3.4.

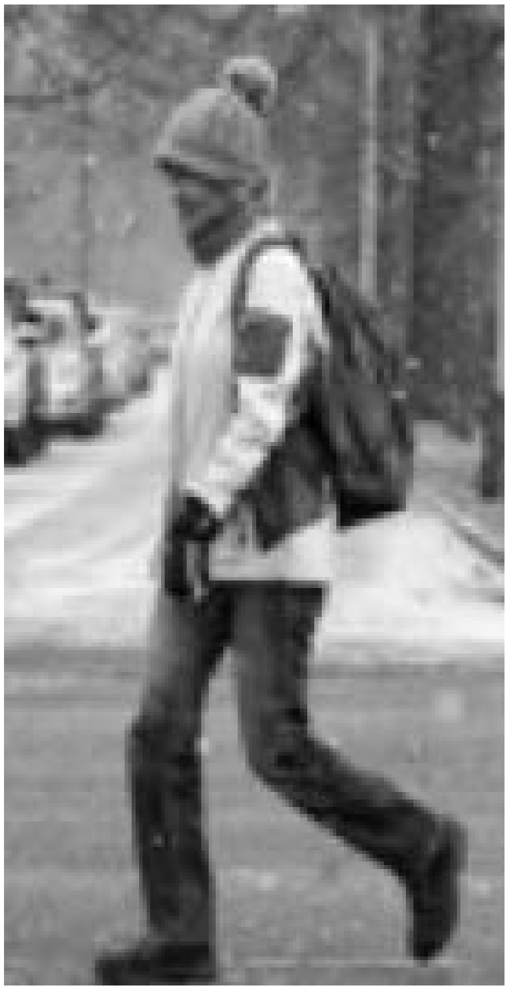

Figure 3.2: Pedestrian image resized to 64×128 pixels. Source: [4]

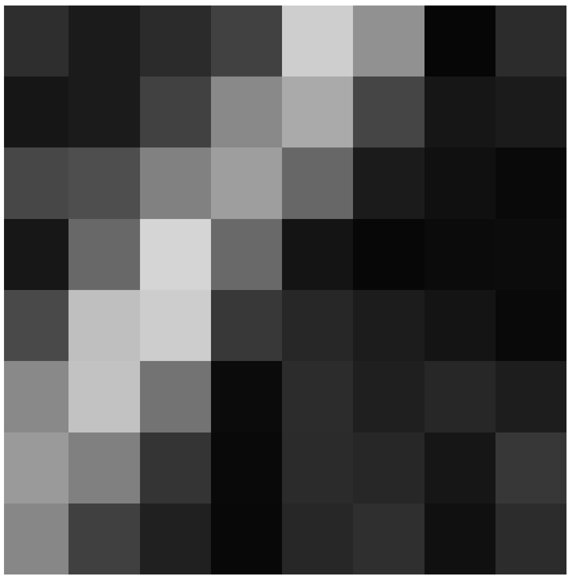

Figure 3.3: 8×8 pixel sample cell taken from the arm of the pedestrian in Figure 3.2

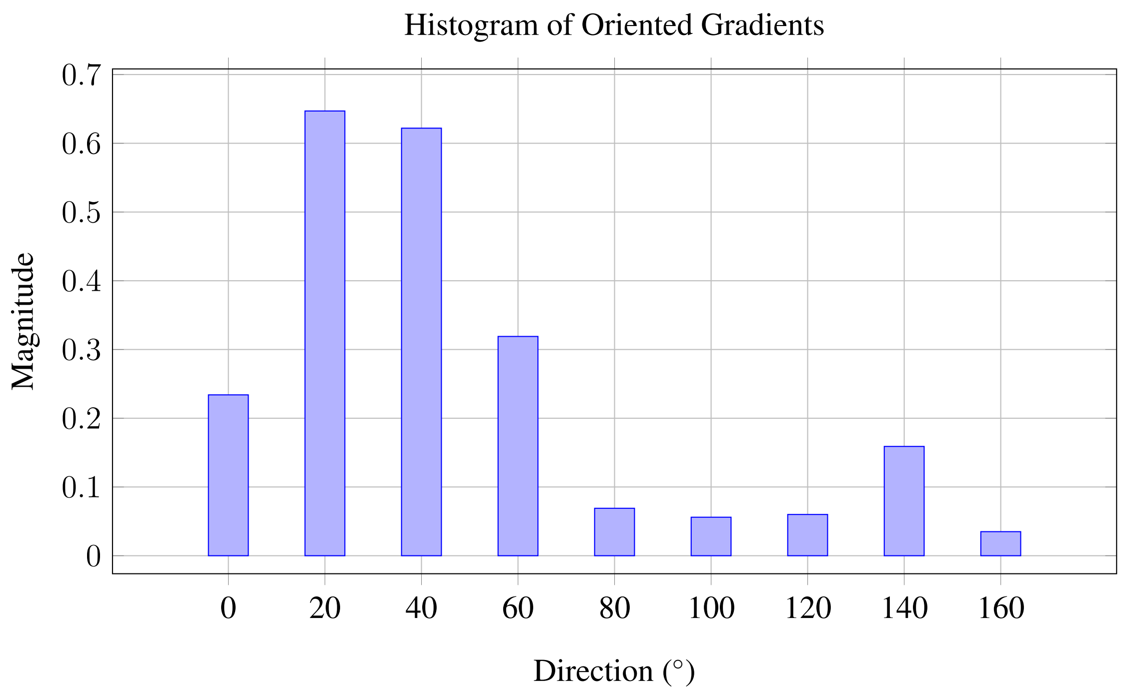

Figure 3.4: Histogram for the image in Figure 3.3.

Analyzing the histogram in Figure 3.4, it indicates that most of the edges in the image sample in Figure 3.3 are in the $20° - 60°$ direction. This aligns with what is visually apparent in the figure. The sample cell in Figure 3.3 shows a piece of the pedestrian's arm, which appears as a diagonal line. The histogram supports this as it indicates that the gradient magnitude is strongest in the diagonal direction. Overall, the process demonstrated in Figures 3.2, 3.3, and 3.4 illustrates how we can reduce an 8×8 cell of 64 pixels into a feature descriptor that contains 9 values.

3.4 Block Normalization

An additional step we used to make our HOG features more resilient to lighting and noise was using block normalization. It works by taking adjacent cells and normalizing the gradient histograms to provide more consistent feature representations.

The first step in block normalization is to define the blocks. Typically, blocks of 4 adjacent cells are used, which means that the block size is 2×2 cells. In our implementation, we used a block size of 2×2 cells, where each cell was 8×8 pixels. The histograms in each cell are then concatenated together to form a single feature vector. In our case, we concatenated 4 cells, each containing 9 bin histograms, so our new feature vector was length 36. Once the concatenated feature vector is created, it is normalized to a unit vector. This calculation is described in Equation 3.6, where $block\ normalized\ vector$ is the normalized concatenated feature vector, $block\ vector$ is the concatenated feature vector, and x_i represents the ith component in the concatenated feature vector.

$$block\ normalized\ vector = \frac{block\ vector}{\sqrt{\sum_i (x_i^2)}} \tag{3.6}$$

This block normalization process is computed for all blocks of cells. Once the computation is completed, the normalized block vectors are concatenated into one final feature vector to represent the image.

3.5 Histogram of Oriented Gradients Application

In general, the purpose of HOG is to obtain a feature vector that can describe the image. The steps outlined in the above sections described the manner in which we computed the HOG features for an image. By applying HOG to thousands of images, we have accumulated a set of HOG features that can describe pedestrians and non-pedestrians.

In the next section, we will discuss the application of these HOG features to create a pedestrian detector. With n-dimensional HOG feature vectors and a support vector machine, we can find the optimal hyperplane to separate pedestrian and non-pedestrian feature vectors.

3.6 Histogram of Oriented Gradients Results

We tested our HOG algorithm on a sample pedestrian and included our results below. The image on the left in Figure 3.5 shows a sample pedestrian. The image in Figure 3.6 shows the HOG features for the pedestrian in Figure 3.5. To compute the HOG features, the image was grayscaled and resized to a size of 64×128 pixels. We used a cell size of 8×8 pixels and a block size of 2×2 cells.

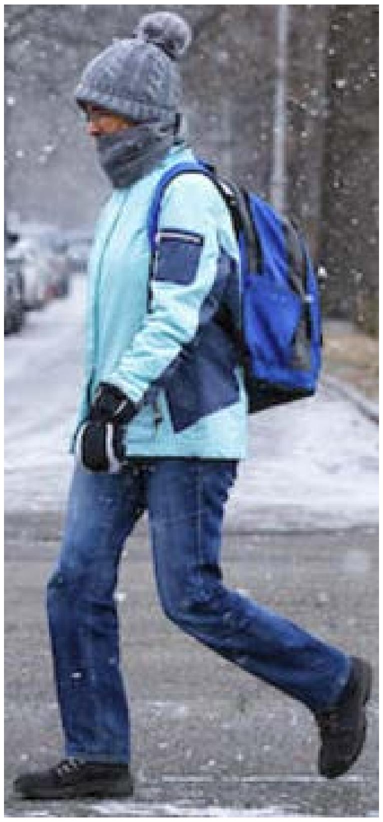
Figure 3.5: Sample pedestrian. Source: [4]

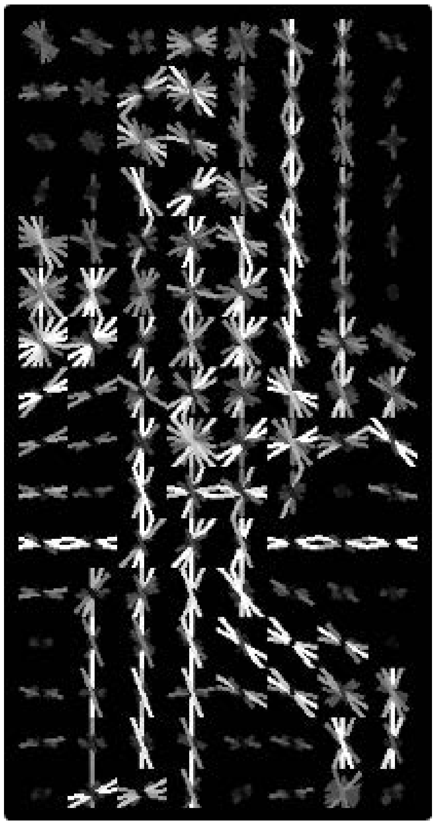
Figure 3.6: HOG features of pedestrian in 3.5.

4 SUPPORT VECTOR MACHINE

Support vector machines (SVMs) are a powerful supervised learning algorithm that constructs an optimal hyperplane to separate data points in an N-dimensional space by maximizing the margin between classes. The hyperplane becomes the decision boundary for the SVM, described by the equation:

$$wx + b = 0 \tag{4.1}$$

The decision boundary is determined by support vectors, which are the data points that lie on the margin boundaries defined by:

$$wx + b = \pm 1 \tag{4.2}$$

For non-linearly separable cases, it implicitly maps the input space into a higher-dimensional feature space. This is done by passing the input vector into a function (called kernel) that outputs a higher dimensional vector allowing the algorithm to find linear decision boundaries in that transformed space, which is called the kernel trick. The optimization problem is commonly solved using quadratic programming techniques where dual formulation involves maximization of the margin and minimization of the classification error, subject to constraints that ensure proper separation of classes [5]. The quadratic programming method is applied for convex optimization only, but there are other loss functions that make the optimization non-convex. Multiple methods can be used, one being an iterative method called gradient descent. We'll explain this later.

The resulting classifier demonstrates remarkable generalization capability and robustness, even in high-dimensional spaces, hence being effective in complex classification tasks.

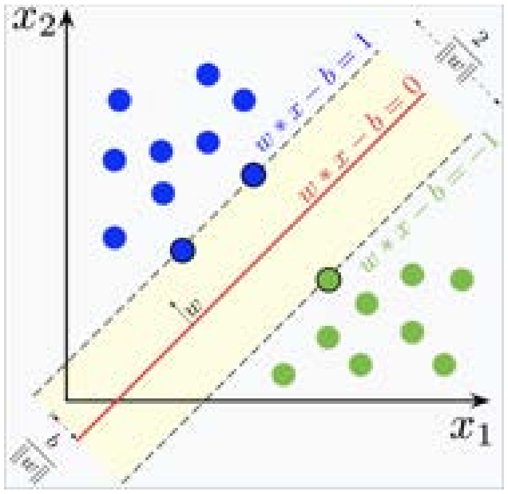

Figure 4.1: Decision boundary with margins. Source: [6]

Support vector machines optimize w and b to fit a line that maximizes the distance between the 2 classes. The support vectors lie on the lines $wx - b = 1$ and $wx - b = -1$.

The width of the margin can be described by:

$$width = \frac{2}{\|w\|} \tag{4.3}$$

The model attempts to maximize this, which can also be interpreted as minimizing $\|w\|$. There are 2 scenarios here: hard margin and margin SVM. We will start by discussing hard margin SVM.

4.1 Hard Margin SVM

Hard margin SVM represents the simplest form of support vector machine classification, applicable only when the data is perfectly separable. In this formulation, the optimization objective focuses solely on maximizing the margin $\frac{2}{||w||}$ while maintaining strict separation between classes, without allowing any misclassifications.

Optimization Problem

Let d_i represent the perpendicular distance of data point i from the hyperplane:

$$d_i = \frac{|wx_i + b|}{\|w\|} \tag{4.4}$$

$$\max d_i = \min_{y_i(wx_i+b)\geq 1} \frac{1}{2}\|w\|^2 \tag{4.5}$$

The Lagrangian for this optimization problem is defined as:

$$L(w, b, \alpha) = \frac{1}{2}\|w\|^2 + \sum_i \alpha_i \left[y_i(wx_i + b) - 1\right] \tag{4.6}$$

subject to:

$$\alpha_i \geq 0, \quad y_i = \text{classification at index } i, \quad y_i(wx_i + b) - 1 = \text{constraint equation.}$$

Taking derivatives of L:

$$\frac{\partial L}{\partial w} = 0, \quad \frac{\partial L}{\partial \alpha} = 0, \quad \frac{\partial L}{\partial b} = 0 \tag{4.7}$$

The gradient conditions yield:

$$\frac{\partial L}{\partial w} = w - \sum_{i=1}^{m} \alpha_i y_i x_i = 0 \quad \implies \quad w = \sum_{i=1}^{m} \alpha_i y_i x_i \tag{4.8}$$

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{m} \alpha_i y_i = 0 \tag{4.9}$$

Substituting these results, the dual form of the Lagrangian is:

$$L(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j (x_i \cdot x_j) \tag{4.10}$$

Here, m is the total number of training samples.

$\frac{1}{2}\|w\|^2 =$ Objective function we want to minimize.

$\alpha_i \geq 0 =$ Lagrangian multipliers.

$y_i =$ Classification at index i.

$y_i(wx_i + b) - 1 =$ Constraint equation.

The objective function simplifies to minimizing:

$$\frac{1}{2}||w||^2 \quad \text{subject to} \quad y_i(w \cdot x_i - b) \geq 1 \tag{4.11}$$

for all training points. This rigid constraint enforces that all data points should be on the correct side of the margin boundaries, hence it is very sensitive to outliers and noise. The major limitation of hard margin SVM is its inability to handle non-linearly separable data, as even a single misclassified point makes the optimization problem infeasible, leading to the development of soft margin SVM variants.

4.2 Soft Margin SVM

Soft margin SVM extends the hard margin approach by introducing the hinge loss function that punishes misclassifications and points that lie too close to the decision boundary. The hinge loss, defined as $\max(0, 1 - y \cdot f(x))$, assigns zero penalty to correctly classified points beyond the margin, while linearly penalizing points based on their distance from the correct side of the margin. This leads to the optimization objective:

$$\min_{w,b} \frac{1}{2}||w||^2 + \sum_{i=1}^{n} \max(0, 1 - y_i(w \cdot x_i - b)) \tag{4.12}$$

This loss function allows the SVM to have larger margins and for points to fall within the margin but penalizes the model for it. This makes soft margin SVM more practical for real-world applications where perfect separation is not achievable, especially in the case of pedestrian detection. This is because the pedestrians can sometimes be very distorted in the camera image and resizing would make it even worse. The representation of such a pedestrian would be closer to the line (the model would be less confident). For such a situation, we need a model that allows points to be within the margin.

In the following section, we'll explain the gradient descent algorithm, its various forms, and methods to iteratively update the parameters w and b of the model to obtain the optimal values.

4.3 The Kernel Trick

4.3.1 Background and Motivation

The kernel trick arises from the need to classify data points that are not linearly separable in their original feature space. Linear models such as support vector machines operate by finding a hyperplane that separates classes. However, in cases where no linear boundary exists, transforming the data into a higher-dimensional space often allows for linear separation. Mathematically, this transformation can be expressed as a mapping:

$$\phi : R^n \to R^m, \quad m > n$$

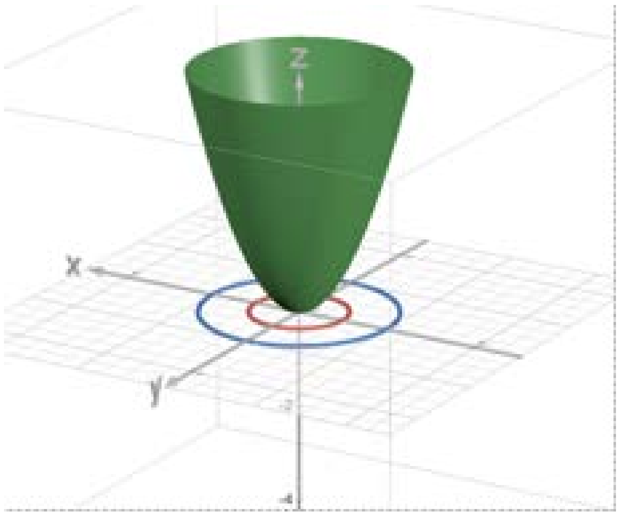

Figure 4.2: Before kernel trick

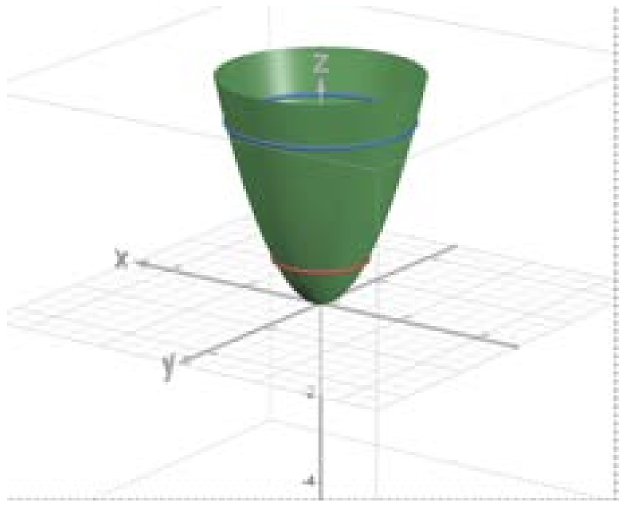

Figure 4.3: After kernel trick

For example, mapping a two-dimensional dataset (e.g., (x, y)) into three-dimensional space by introducing a new dimension $z = x^2 + y^2$ can allow for the construction of a linear separator, as illustrated in the accompanying diagram with concentric circles. However, explicitly computing this transformation for high-dimensional spaces is computationally expensive, which is where the kernel trick plays a pivotal role.

4.3.2 Mathematical Foundation

At the core of the kernel trick lies the observation that many machine-learning algorithms rely on: the dot product between feature vectors. For instance, in SVMs, the decision boundary depends on:

$$f(x) = \sum_{i=1}^{m} \alpha_i y_i (\phi(x_i) \cdot \phi(x)) + b \tag{4.13}$$

where:

x_i = Training data points

y_i = Corresponding labels ($+1$ or -1)

α_i = Lagrange multipliers determined during optimization

$\phi(x)$ = Feature transformation function (kernel function)

Instead of explicitly computing $\phi(x)$, the kernel trick leverages a kernel function $k(x_i, x_j)$, which computes the dot product in the higher-dimensional space directly:

$$k(x_i, x_j) = \phi(x_i) \cdot \phi(x_j) \tag{4.14}$$

By replacing the dot product $\phi(x_i) \cdot \phi(x_j)$ with $k(x_i, x_j)$, one can operate as though the transformation $\phi(x)$ has been performed, without ever explicitly computing it. This bypasses the computational expense of directly handling high-dimensional feature vectors.

4.3.3 Common Kernel Functions

Several kernel functions are widely used, each corresponding to a specific transformation $\phi(x)$:

- Linear Kernel:

$$k(x_i, x_j) = x_i \cdot x_j \tag{4.15}$$

  This corresponds to the original feature space, with no transformation.

- Polynomial Kernel:

$$k(x_i, x_j) = (x_i \cdot x_j + c)^d \tag{4.16}$$

  This maps the data into a space where interactions up to degree d are considered.

- Radial Basis Function (RBF) or Gaussian Kernel:

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right) \tag{4.17}$$

  This corresponds to an infinite-dimensional transformation and is especially effective in capturing localized relationships between data points.

- Sigmoid Kernel:

$$k(x_i, x_j) = \tanh(\alpha x_i \cdot x_j + c) \tag{4.18}$$

  This is inspired by neural network activation functions and is useful in certain contexts.

Each kernel implicitly defines a specific geometry in the transformed space, enabling the model to learn non-linear decision boundaries.

4.4 Gradient Descent

Another way to optimize the loss function for our SVM is the gradient descent algorithm. This method is often used to solve optimization problems with non-convex loss functions: functions that have multiple local minima and maxima. Although the hinge loss gives a convex function to optimize: a function with 1 absolute minimum, multiple loss functions like ramp loss or truncated hinge loss can give a non-convex loss function.

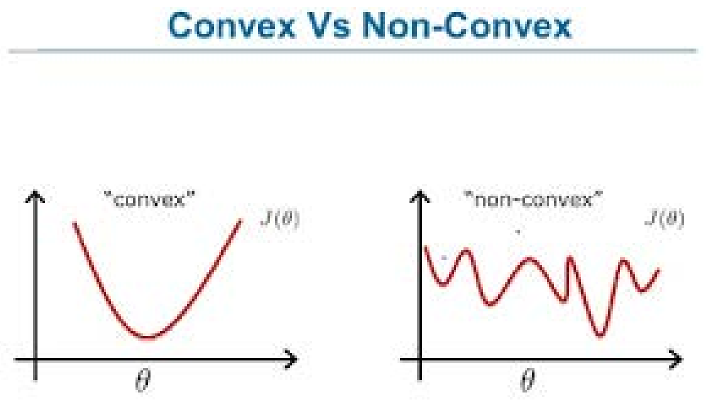

Figure 4.4: Convex vs Non-convex. Source: [7]

Gradient descent usually drives the parameters to very low values near the minima without any guarantee of reaching the absolute minima.

The gradient of a function $f(x)$ at a certain point gives the direction of change in x that will result in the highest rate of increase in $f(x)$. So if we take the negative of the gradient, it would give us the direction of x which would result in the steepest decrease in $f(x)$.

So, if we take a small step (ϵ) in the direction opposite to the gradient we'll reduce the function $f(x)$ the fastest. The gradient will then be recalculated and another step will be taken. This approach will continue till a local minimum is reached.

In the paragraph above, ϵ is also called the step size or (more commonly) the learning rate. The equation for gradient descent is:

$$x_{new} = x_{old} - \alpha \nabla f(x) \tag{4.19}$$

Here, α is the step size (learning rate), while x represents the parameters of the SVM.

There are 3 ways of implementing gradient descent:

- Stochastic gradient descent
- Mini-batch gradient descent

- Batch gradient descent

4.4.1 Batch Gradient Descent

Batch Gradient Descent (BGD) is a fundamental optimization algorithm used in machine learning to minimize the cost function of a model. It is particularly useful for finding the optimal parameters of (w and b) SVMs.

Principle The core principle of BGD is to iteratively adjust the model parameters in the direction that reduces the cost function most rapidly. This direction is determined by calculating the gradient of the cost function with respect to each parameter, using the entire training dataset in each iteration. For calculating the gradient using the entire dataset. The gradient is calculated for every data point fed into the model and added to a variable. This variable at the end will store the sum of all the gradients. The sum is then divided by the number of data points to obtain the average gradient of the entire dataset.

Algorithm The BGD algorithm can be summarized as follows:

1. Initialize the model parameters randomly or with some predefined values.
2. Calculate the gradient of the cost function using the entire training dataset.
3. Update the parameters by moving them in the opposite direction of the gradient.
4. Repeat steps 2 and 3 until convergence or a maximum number of iterations is reached.

Mathematically, the parameter update rule can be expressed as:

$$w := w - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial w} f(y; x^{(i)}, y^{(i)}) \tag{4.20}$$

where:

w = a parameter of the model

α = the learning rate

$f(y)$ = the cost function

n = the number of data points.

$\frac{\partial}{\partial w} J(w)$ = the partial derivative of the loss function with respect to w

Since batch gradient descent takes the average gradient over the entire dataset, it is guaranteed that the step taken in the update of the parameters (w) will reduce the sum of the loss over the entire dataset. This means that an update is guaranteed to improve the performance of the SVM on the entire dataset as a whole.

Advantages

- Stability: BGD provides a stable convergence path due to the use of the entire dataset in each iteration.
- Guaranteed convergence: For convex cost functions, BGD is guaranteed to converge to the global minimum.

- Deterministic: The algorithm produces the same results for the same initial conditions, making it easier to debug and reproduce results. This is because its noisy updates help it to jump out of local minima.

Disadvantages

- Computational cost: BGD can be computationally expensive for large datasets, as it requires the entire dataset to be processed in each iteration.
- Memory intensive: For very large datasets, storing the entire dataset in memory may not be feasible.
- Slow convergence: In some cases, BGD may converge slowly, especially when the cost function is very flat near the optimum.

4.4.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a variation of gradient descent that updates model parameters using a single randomly selected training example in each iteration, making it more efficient for large-scale learning problems. If the dataset has k points the gradient will be calculated and the parameters will be updated k times.

Principle Unlike batch gradient descent, SGD approximates the true gradient by computing it for a single randomly selected training example. This introduces noise into the optimization process but often leads to faster convergence and better generalization.

The better generalization is due to the noise added to the gradients. Because of this noise, the updates are more random and take into consideration every single data point. This helps the model prevent overfitting the data by considering the inherent noise in every data point. It also helps the model in finding flatter minima than very sharp minima. A sharp minimum could correspond to the w and b that work exceptionally well for the training data which implies that they might not generalize well to the real-world data. [8]

Algorithm The SGD algorithm follows these steps:

1. Initialize model parameters randomly
2. Randomly shuffle the training dataset
3. For each training example:
    - Calculate the gradient using only this example
    - Update the parameters immediately
4. Repeat steps 2-3 for multiple epochs until convergence

The parameter update rule for SGD is:

$$w := w - \alpha \frac{\partial}{\partial w} f(y; x^{(i)}, y^{(i)}) \tag{4.21}$$

where:

$(x^{(i)}, y^{(i)})$ = a single training example

w_j = the j-th parameter

α = the learning rate

$f(y; x^{(i)}, y^{(i)})$ = the loss for a single data point

Advantages

- Memory efficient: Processes one example at a time
- Fast updates: More frequent parameter updates
- Ability to escape local minima: Random noise helps avoid poor local minima. This is because the noisy updates [9]

Disadvantages

- High variance: Updates can be noisy due to using single examples
- No guarantee of exact convergence: May oscillate around the optimal point
- Requires careful learning rate tuning: More sensitive to learning rate selection

4.4.3 Mini-Batch Gradient Descent

Mini-batch gradient descent combines the best of both worlds: batch and stochastic gradient descent. In practice, this is a very practical compromise between computation efficiency and convergence stability. Hence it lies somewhere in between batch and stochastic gradient descent.

The algorithm updates parameters using small, random subsets (mini-batches) of the training data, typically ranging from 32 to 256 examples per batch. This gives more stable convergence compared to SGD but still offers computational efficiency.

The steps for mini-batch gradient descent are:

1. Initialize model parameters randomly
2. Divide the training dataset into mini-batches

3. For each mini-batch:

    - Calculate the average gradient over the mini-batch
    - Update the parameters using this gradient

4. Repeat steps 2-3 for many epochs

The parameter update rule becomes:

$$w := w - \alpha \frac{1}{b} \sum_{i=1}^{b} \frac{\partial}{\partial w} f(y; x^{(i)}, y^{(i)}) \tag{4.22}$$

where:

b = the mini-batch size

$\sum_{i=1}^{b}$ = the sum over examples in the mini-batch

f = the loss function

Advantages

- Balanced computation: More efficient than batch gradient descent
- Reduced variance: More stable than SGD
- Memory efficient: Requires less memory than batch gradient descent
- Hardware consideration: Many machine learning libraries and hardware have optimizations for mini-batches, which make it fast.

Disadvantages

- Batch size selection: Requires tuning of an additional hyperparameter
- Memory management: Needs efficient implementation for batch processing
- Learning rate sensitivity: Still requires careful learning rate selection

Choosing Batch Size The selection of mini-batch size involves several considerations:

- Memory constraints: Must fit in available computing resources
- Training stability:
  - Larger batches provide more stable gradients
  - Smaller batch sizes will add more noise and can improve generalization.

4.4.4 Updated Rule for SVM

For gradient descent, we require a cost function that measures the error in our model. For the purposes of the paper, we will be using the hinge loss since we want to make a soft margin SVM. The hinge loss is defined as follows:

$$L(y, f(x)) = \max(0, 1 - y \cdot f(x)) \tag{4.23}$$

Here, $f(x)$ represents the predicted value for the input x. $f(x) = 0$ is the linear decision boundary.

$$f(x) = wx - b \tag{4.24}$$

The optimization aims to achieve the best possible value of w and b. Here, the width of the margin in the SVM is:

$$width = \frac{2}{\|w\|} \tag{4.25}$$

so we need to minimize the value of $\|w\|^2$. Additionally, we want to minimize the hinge loss. So the final loss function for n points becomes:

$$J = \lambda\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i(w \cdot x_i - b)) \tag{4.26}$$

$$J_i = \begin{cases} \lambda\|w\|^2 & \text{if } y_i \cdot f(x) \geq 1 \\ \lambda\|w\|^2 + 1 - y_i(w \cdot x_i - b) & \text{else} \end{cases} \tag{4.27}$$

Here λ is a parameter set by the person. It quantifies how much importance should be given to minimizing $\|w\|$ compared to minimizing the hinge loss.

$y_i \cdot f(x) \geq 1$ refers to the situation of the SVM making a correct prediction and the point is outside the margin of $\frac{2}{\|w\|}$.

On taking the gradients of these equations with w and b. We get the following cases:

$$\frac{dJ_i}{dw_k} = \begin{cases} 2\lambda w_k & \text{if } y_i \cdot f(x) \geq 1 \\ 2\lambda w_k - y_i \cdot x_{ik} & \text{else} \end{cases} \tag{4.28}$$

$$\frac{dJ_i}{db} = \begin{cases} 0 & \text{if } y_i \cdot f(x) \geq 1 \\ y_i & \text{else} \end{cases} \tag{4.29}$$

The updated rule for the parameters of the SVM has 2 cases, assuming a mini-batch gradient descent approach with a batch size of b:

$$w = \begin{cases} w - \frac{1}{b}\sum_{i=1}^{b} 2\lambda w_k & \text{if } y_i \cdot f(x) \geq 1 \\ w - \frac{1}{b}\sum_{i=1}^{b}(2\lambda w_k - y_i \cdot x_{ik}) & \text{else} \end{cases} \tag{4.30}$$

$$b = \begin{cases} b & \text{if } y_i \cdot f(x) \geq 1 \\ b - \frac{1}{b}\sum_{i=1}^{b} y_i & \text{else} \end{cases} \tag{4.31}$$

4.5 Training

When utilizing an SVM in our project, the way of extracting both positive (pedestrian) and negative (non-pedestrian) samples from each image in the dataset greatly influenced the accuracy of our model. We will discuss the methods we used for both classes.

4.5.1 Positive Samples

The images in the dataset contain ground truths for each pedestrian, visualized in Figure 4.5.

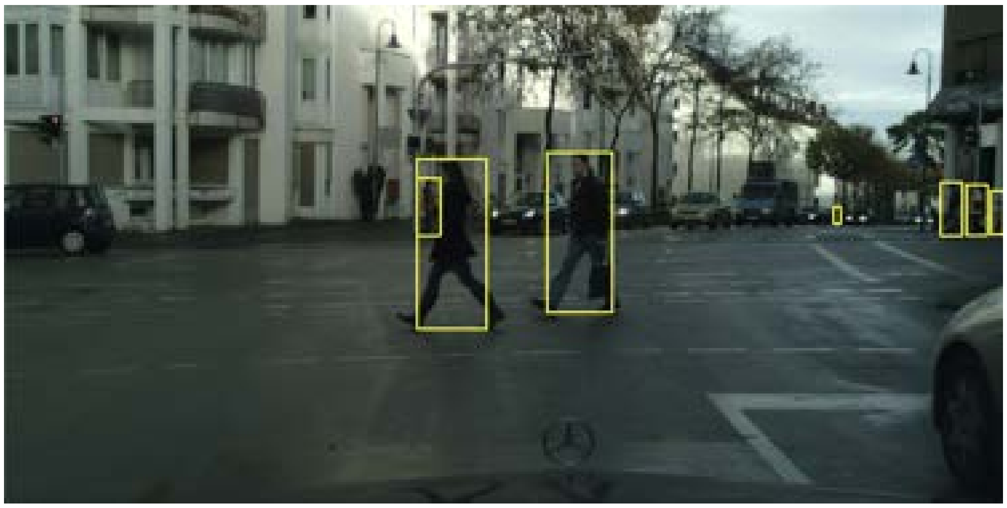
Figure 4.5: Ground truths of a training image

We can extract each bounding box with a size greater than 32×64 pixels, which ensures that the features of our training data are well-defined and do not contain a disproportionate amount of noise. Afterward, we resize the image to 64×128 and extract its HOG features.

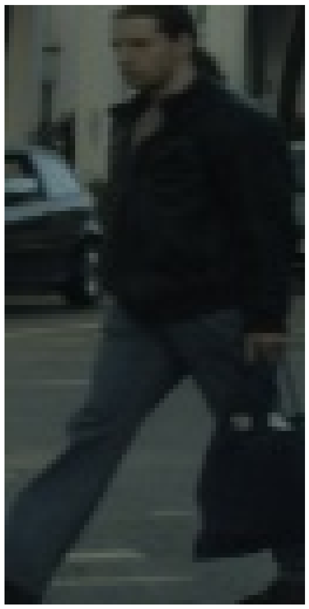
Figure 4.6: Positive sample from Figure 4.5 resized to 64×128 pixels

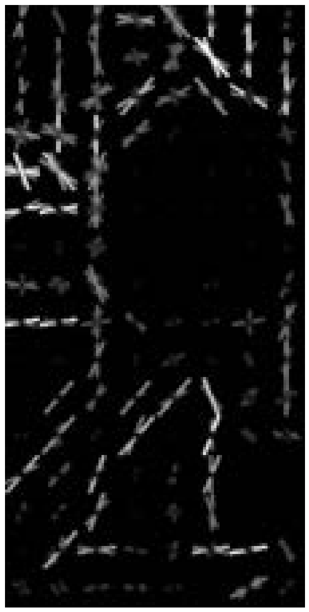
Figure 4.7: HOG of sample in Figure 4.6 using 8×8 pixel cells

4.5.2 Negative Samples

The Cityscapes dataset [2] did not provide specific annotations for regions without pedestrians, thus, we devised our own way of obtaining negative samples. We want the samples to be generated randomly in terms of both location and size to ensure the model includes a diverse range of images. For our model, we will only generate boxes with a $1 : 2$ aspect ratio and a width between 32 and 256 pixels. This is also under the assumption that each image's dimension is greater than or equal to 256×512 pixels.

To achieve this, let each image have a width of W and a height of H. Additionally, let $X_1, X_2, X_3 \sim U(0, 1)$ denote random float variables uniformly distributed between 0 and 1 (inclusive). Now, we can use the following equations to find the width, height, and x and y coordinates of a randomly generated box within the image:

$$w = [(256 - 32) \cdot X_1 + 32] \tag{4.32}$$

$$h = w \cdot 2 \tag{4.33}$$

$$x = [(W - w) \cdot X_2] \tag{4.34}$$

$$y = [(H - h) \cdot X_3] \tag{4.35}$$

where:

w = width of box in pixels

h = height of box in pixels

x = x coordinate of box in pixels

y = y coordinate of box in pixels

Our model draws 4 negative samples from each image. If any generated sample overlaps with a pedestrian bounding box, which is known from the ground truths, we will attempt to regenerate the sample until there is no overlap. To calculate the overlap between boxes, refer to Equation 5.7.

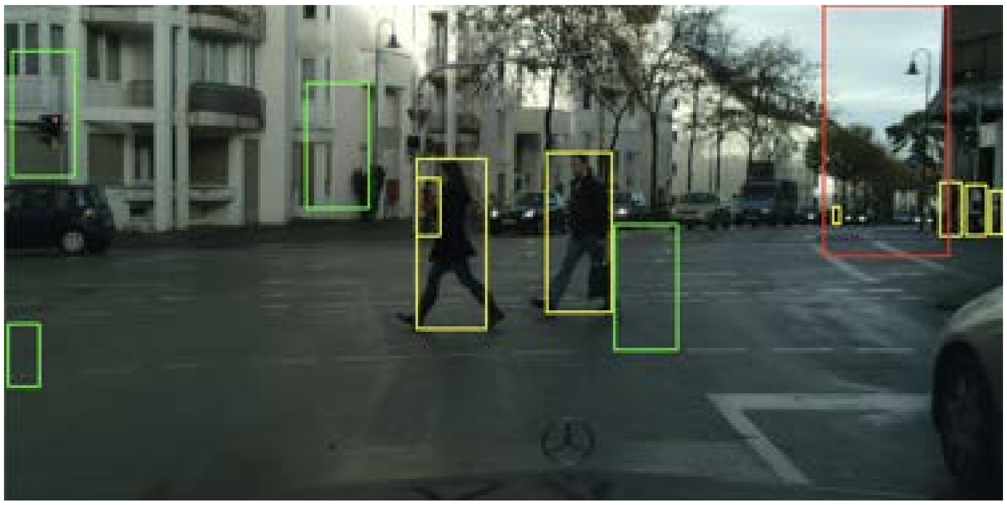

Figure 4.8: 4 random negative samples from image with width of W and height of H. Ground truths are drawn in yellow, successfully generated negative samples are drawn in green, and overlapping generated negative samples are drawn in red.

After extracting each sample, we will resize it to 64×128 pixels and extract its HOG features.

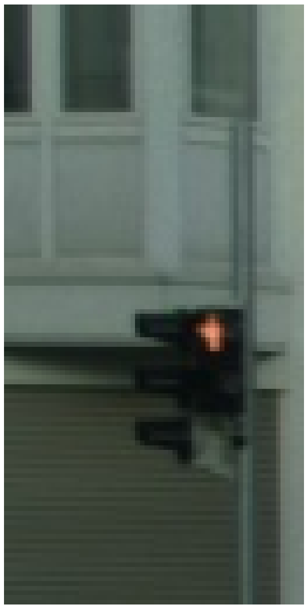

Figure 4.9: Negative sample from Figure 4.8 resized to 64×128 pixels

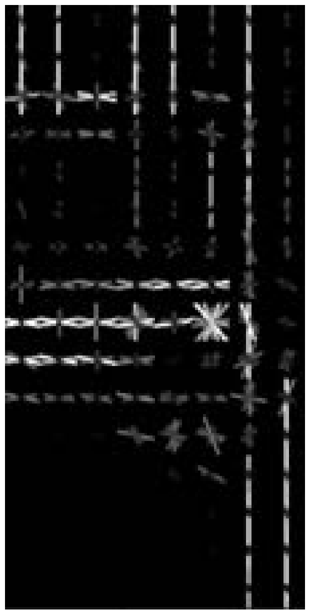

Figure 4.10: HOG of sample in Figure 4.9 using 8×8 pixel cells

4.5.3 Hard Negative Mining

To boost the performance of our SVM model, we incorporated a technique known as hard negative mining, which specifically targets the reduction of false positive detections (detecting a pedestrian when there is none). The idea behind the method is that by selecting negative images that the model incorrectly identifies as positive and then retraining the model with said images, the model is forced to focus on images it struggles with, therefore drawing a decision boundary that more accurately separates distinguishing features between the two classes. In our case, the following steps would be applied in order:

1. Train an initial model on a portion of the dataset.

2. Test the model on exclusively non-pedestrian images taken from another portion of the dataset.

3. For every false positive classification, add the image to a list.

4. When finished testing, retrain the initial model with the additional list of false positives.

4.5.4 Half-Pedestrian and Shifted-Box Training

While sliding window technique is effective for detecting pedestrians, it faces challenges in accurately capturing the entire pedestrian, especially when the window only partially captures the pedestrian. Since this can cause false negatives or missed detections, we incorporated special training algorithms: half-pedestrian training and shifted-box training.

Half-Pedestrian Training Half-Pedestrian Training involves creating more positive samples from the dataset by splitting the ground truth bounding boxes into 2 halves: top and bottom. This lets the model not only recognize the full pedestrian but also their partial features.

For a ground truth box $B = (x, y, w, h)$, the half-pedestrian boxes are defined as:

- Top Half: $B_{top} = (x, y, w, \frac{h}{2})$

- Bottom Half: $B_{bottom} = (x, y + \frac{h}{2}, w, \frac{h}{2})$

Results Half-Pedestrian training results

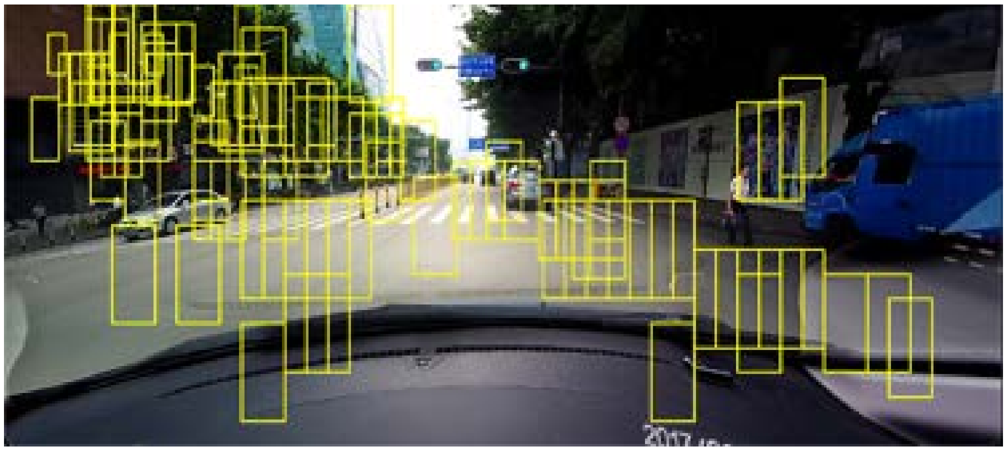

Figure 4.11: Old Model ($threshold = 0.75$)

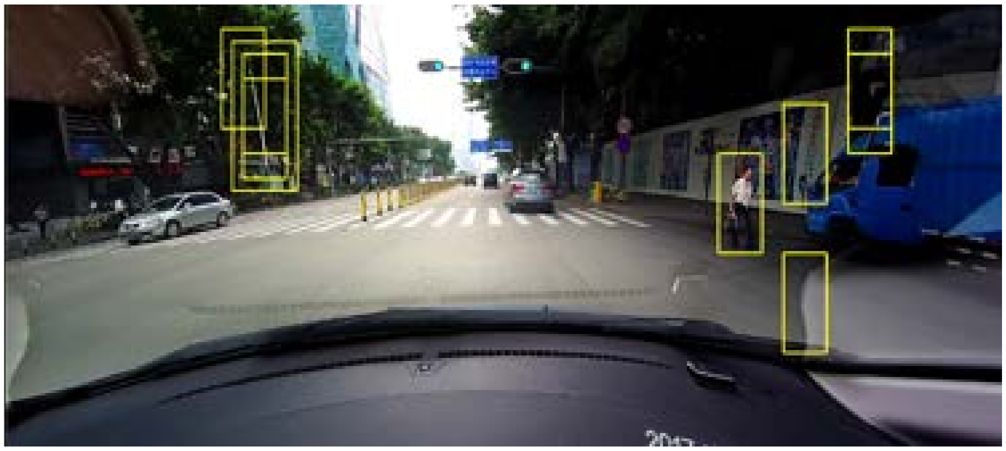

Figure 4.12: Model with Half-Pedestrian Training ($threshold = 0.75$)

Shifted-box Training Shifted-box training is another algorithm that further enhances the robustness of our detection system. It does so by introducing shifts to the ground truth boxes. For each ground truth box $B = (x, y, w, h)$, the algorithm shifts the box by randomly adjusting the coordinates x and y while maintaining the original dimensions w and h.

$$B_{\text{shifted}} = (x + \Delta x, y + \Delta y, w, h), \quad \Delta x, \Delta y \in (-\delta, \delta) \tag{4.36}$$

In the equation above, δ is the maximum shift allowed so that parts of the pedestrians are still in the window. The shifted boxes simulate variations in pedestrian localization due to imperfect bounding box placement during sliding window.

Results Shifted-box training results

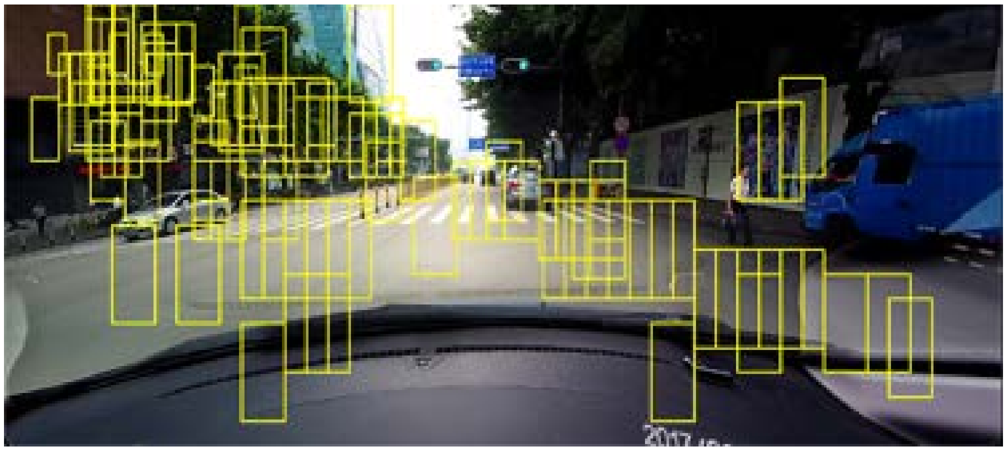

Figure 4.13: Old Model ($threshold = 0.75$)

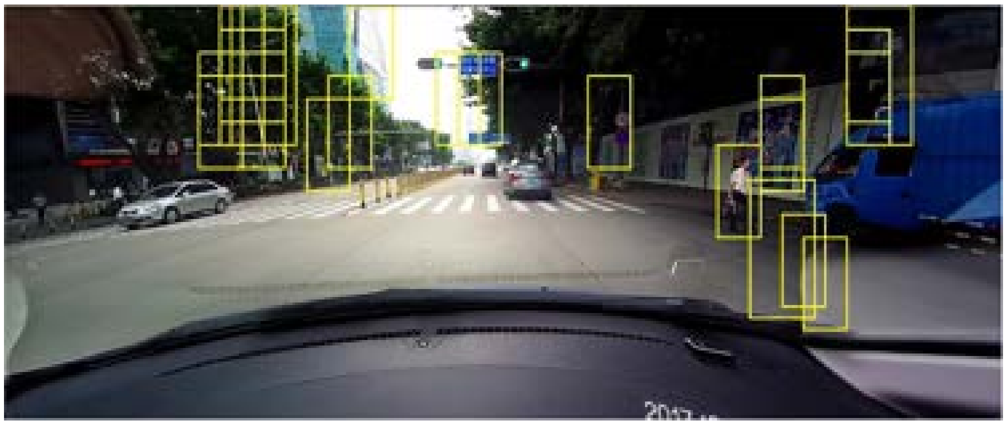

Figure 4.14: Model with Shifted-Box Training ($threshold = 0.75$)

5 SLIDING WINDOW

Sliding window detection is an extremely popular approach in computer vision for object detection tasks, including pedestrian detection. The technique involves systematically scanning an image with windows of predefined sizes, extracting feature vectors for each window, and classifying them using a trained classifier, such as a support vector machine. Unlike classification tasks, where objects are already segmented, object detection requires locating objects within an image and determining their boundaries. By scanning the image at every possible location and scale, this method, when combined with histograms of oriented gradients for feature extraction, is particularly effective for pedestrian detection.

5.1 Mathematics

Given an input image I of dimensions $H \times W$, the sliding window is applied to a grid of coordinates (x, y) with a fixed window size $w \times h$. Let the set of all possible window coordinates be:

$$\mathcal{W} = \{(x, y) \mid x \in [0, W - w],\, y \in [0, H - h]\} \tag{5.1}$$

The window is moved across the image with a step size s, where smaller step sizes increase computational cost, but allow us to detect smaller pedestrians. The issue of greater computational cost is addressed through threading later in this paper.

Each window $w_{(x,y)} \in \mathcal{W}$ is passed through the HOG feature extractor to produce a feature vector $f_{(x,y)}$:

$$f_{(x,y)} = \text{HOG}(w_{(x,y)}) \tag{5.2}$$

HOG computes gradients and orientation histograms and normalizes these features to encode information on texture and shape critical to pedestrian detection.

The SVM classifier assigns a confidence score $c(f_{(x,y)})$ to each feature vector $f_{(x,y)}$. The confidence score is given by:

$$c(f_{(x,y)}) = \frac{w^\top f_{(x,y)} + b}{\|w\|} \tag{5.3}$$

where w and b are the hyperplane parameters learned during SVM training. A window is classified as containing a pedestrian if:

$$c(f_{(x,y)}) \geq 0 \tag{5.4}$$

5.2 Image Resizing Technique

An alternate technique we tried implementing is the sliding window with image resizing [10]. In this approach, rather than using multiple window sizes to detect pedestrians, the input image is resized multiple times. A fixed window size $w \times h$ is then applied to each resized version of the image. This method can significantly reduce computational cost as it avoids the need to repeatedly extract features from windows of varying sizes.

Given an image I of dimensions $H \times W$, a scaled version of the image I_s is created. The image is resized with a scaling factor s, where $s \in S$ and S is a set of scales.

$$I_s = \text{Resize}(I, s \cdot H, s \cdot W), \quad s \in \mathcal{S} \tag{5.5}$$

For each scaled image, I_s, the sliding window is applied with the fixed size $w \times h$. Then the feature vector $f^s_{(x,y)}$ for each window $w^s_{(x,y)}$ is extracted using HOG.

$$f^s_{(x,y)} = \text{HOG}(w^s_{(x,y)}) \tag{5.6}$$

The SVM classifier then predicts the confidence score $c(f^s_{(x,y)})$ for each window as described before.

The image resizing technique is computationally efficient as it avoids handling multiple window sizes. However, it introduces challenges such as errors in re-mapping detection coordinates back to the original image due to scaling and rounding, which can affect the accuracy of bounding boxes. In addition, image down-scaling can result in loss of detail, leading to missed detections for small or distant pedestrians. Balancing efficiency and accuracy requires careful selection of scaling factors and post-processing steps.

5.3 Threading

In our implementation of the sliding window operation, we observed that as we reduced the step size, the number of iterations required to scan the entire image increased. This directly resulted in longer runtime, which can be problematic for applications that need to process images or videos in real time or with large datasets. We had to find a way to optimize our current operation and we figured that the best way to accomplish this was to bring concurrency into our program and allow the sliding windows to run simultaneously. However, the question was how many threads should we utilize for the most efficient program.

- $WSF =$ Window Size Format $= (w, h, s)$
- $w =$ width of window size (pixels)
- $h =$ height of window size (pixels)
- $s =$ step size (pixels)
- t(WSF) $=$ time till completion of WSF (mins)
- $T =$ total time (mins)
- n(WSF) $=$ threads for WSF

No Threads

$$(64, 128, 32) \rightarrow (32, 64, 16) \rightarrow (24, 48, 12) \rightarrow (16, 32, 8)$$

$$T \approx 40$$

Four Threads

$$\text{1 Thread} - (64, 128, 32)$$

$$\text{1 Thread} - (32, 64, 16)$$

$$\text{1 Thread} - (24, 48, 12)$$

$$\text{1 Thread} - (16, 32, 8)$$

$$T \approx 20$$

$$t(64,128,32) < t(32,64,16) < t(24,48,12) < t(16,32,8) \approx T \approx 20$$

Optimization

$$n(\text{WSF}) \propto \frac{1}{s}, \quad n(\text{WSF}) = \frac{k}{s} \text{ where k is some constant.}$$

$$\text{Ex. } k = 32 : n(64,128,32) = 1,\ n(32,64,16) = 2, \ldots$$

This allows us to dynamically increase the number of threads according to the step size of the sliding window, enabling our program to run at extremely fast speeds. As of currently, we have not implemented dynamic threading and are using one thread per window operation, but we look to implement dynamic threading as we move from images to videos.

5.4 Post-Processing

Post-processing in sliding window detection is critical for refining the initial outputs of the detection algorithm, ensuring both accuracy and efficiency. Techniques such as thresholding, non-maximum suppression, and color filtering help drastically reduce the number of false positives in our detections and leave us with an accurate model to use for crash avoidance.

5.4.1 Thresholding

Thresholding plays a vital role in sliding window detection by reducing the number of false detections based on the classifier's confidence in its predictions. As of right now, we are accepting any detection with a positive confidence score to contain a pedestrian. However, by selecting an appropriate threshold T, windows that produce low confidence scores can be filtered out, significantly reducing false positive detections in non-pedestrian regions, such as fences, trees, and signs. Through a trial-and-error method, we were able to deduct a threshold value that would minimize false positives while avoiding false negatives.

Thresholding Results In Figure 5.1, it can be seen that our model not only captures the pedestrians, but also many of the surrounding objects such as railings, the hood ornament, and general background noise. However, after simply applying a threshold of $T = 1.2$, the number of detections goes down significantly as shown in Figure 5.2.

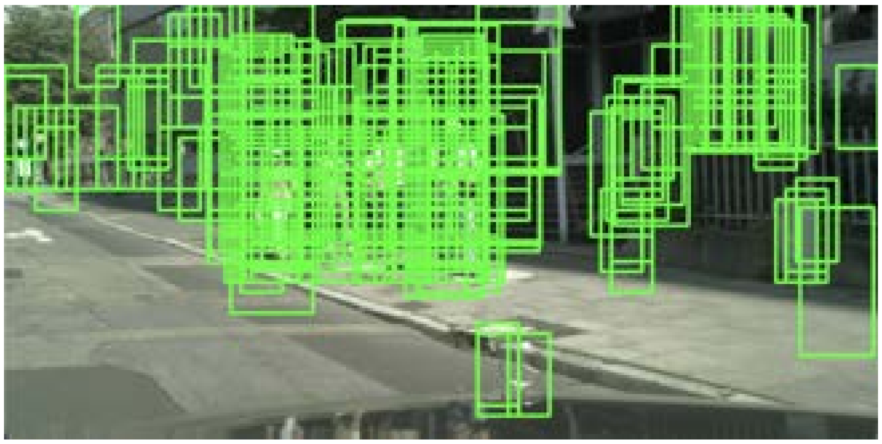

Figure 5.1: Pedestrian detection output before thresholding

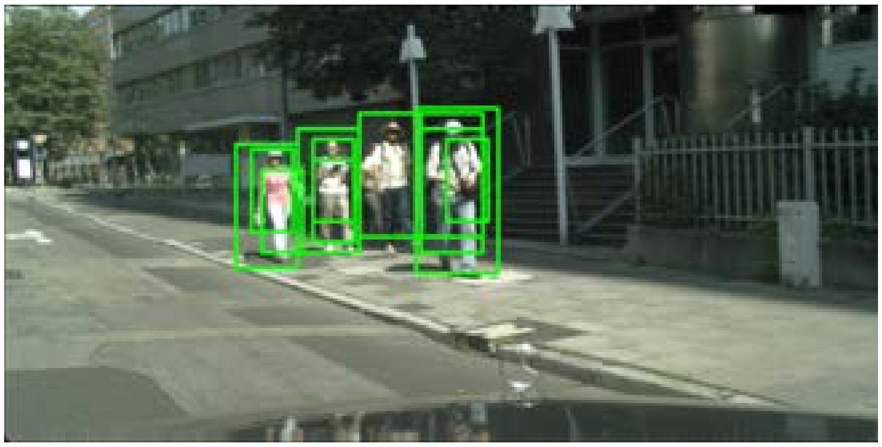

Figure 5.2: Pedestrian detection output after thresholding with $T = 1.2$

5.4.2 Non-Maximum Suppression

Non-Maximum Suppression (NMS) is a crucial post-processing technique that can be used to remove redundant or overlapping bounding boxes that may have been created during our sliding window operation. Without NMS, multiple bounding boxes may be predicted around the same pedestrian due to overlapping features or multiple sliding windows detecting the same object.

Given a set of N predicted bounding boxes $\mathcal{B} = \{B_1, B_2, \ldots, B_N\}$, each associated with:

- A confidence score $S = \{s_1, s_2, \ldots, s_N\}$, where $s_i \in [0, 1]$.
- A set of coordinates $B_i = (x_i^{(1)}, y_i^{(1)}, x_i^{(2)}, y_i^{(2)})$, defining the top-left $(x_i^{(1)}, y_i^{(1)})$ and bottom-right $(x_i^{(2)}, y_i^{(2)})$ corners.

Our goal is to select a subset of bounding boxes $\mathcal{B}' \subseteq \mathcal{B}$, such that the selected boxes have minimal overlap with each other and the subset preserves the most confident predictions. In order to achieve this, we will need to utilize the Intersection over Union (IoU).

Compute Intersection over Union (IoU) For any two bounding boxes B_i and B_j, the IoU is defined by equation 5.7.

$$\text{IoU}(B_i, B_j) = \frac{\text{Area}(B_i \cap B_j)}{\text{Area}(B_i \cup B_j)} \tag{5.7}$$

To calculate the intersection of two bounding boxes, as shown in Figure 5.3, we must calculate the area of the overlapping region, which is defined by Equation 5.8.

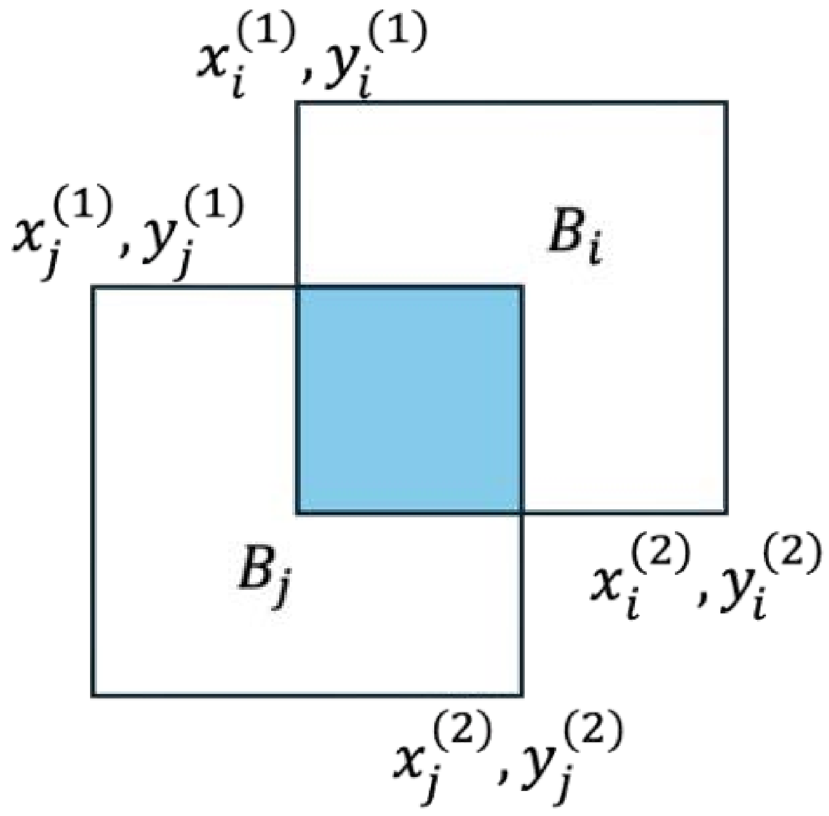

Figure 5.3: Intersection of two bounding boxes B_i and B_j

$$\text{Area}(B_i \cap B_j) = (\min(x_i^{(2)}, x_j^{(2)}) - \max(x_i^{(1)}, x_j^{(1)})) \cdot (\min(y_i^{(2)}, y_j^{(2)}) - \max(y_i^{(1)}, y_j^{(1)})) \quad (5.8)$$

Then, we can use this information to calculate the union, shown in Figure 5.4, of the two bounding boxes. To do this, we must calculate the areas of the two bounding boxes B_i and B_j and subtract the area of the intersection between B_i and B_j. This is given by Equation 5.9.

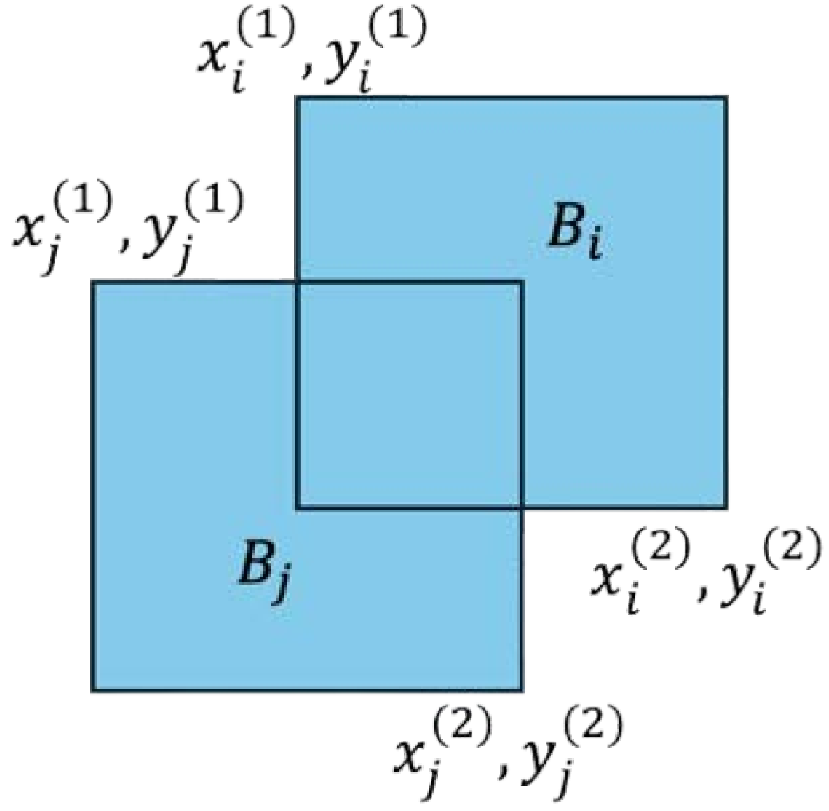

Figure 5.4: Union of two bounding boxes B_i and B_j

$$\text{Area}(B_i \cup B_j) = \text{Area}(B_i) + \text{Area}(B_j) - \text{Area}(B_i \cap B_j) \tag{5.9}$$

The area of each bounding box B_i is computed as:

$$\text{Area}(B_i) = \left(x_i^{(2)} - x_i^{(1)}\right) \cdot \left(y_i^{(2)} - y_i^{(1)}\right) \tag{5.10}$$

Therefore, the equation for the union is:

$$\text{Area}(B_i \cup B_j) = \left(x_i^{(2)} - x_i^{(1)}\right) \cdot \left(y_i^{(2)} - y_i^{(1)}\right) + \left(x_j^{(2)} - x_j^{(1)}\right) \cdot \left(y_j^{(2)} - y_j^{(1)}\right)$$
$$- (\min(x_i^{(2)}, x_j^{(2)}) - \max(x_i^{(1)}, x_j^{(1)})) \cdot (\min(y_i^{(2)}, y_j^{(2)}) - \max(y_i^{(1)}, y_j^{(1)})). \tag{5.11}$$

Now that we have defined the equation for IoU, Equation 5.7, we must follow the NMS steps to remove redundant or overlapping bounding boxes.

Procedure The NMS steps proceed as follows:

$$\mathcal{B}' = \emptyset$$

$$\mathcal{B} \leftarrow \text{Sort}(\mathcal{B}, \text{by } S \text{ in descending order})$$

Traverse down $\mathcal{B}$, computing the IoU of each box with the boxes further down the list.

If the IoU is greater than a threshold T, remove the box with the lower S.

Results Implementing NMS into our pedestrian detector drastically reduced the number of redundant bounding boxes in our images, as can be seen in the difference between Figure 5.5 and Figure 5.6. NMS ensured that the displayed bounding boxes were clear and non-redundant, making it easier for users to interpret the output.

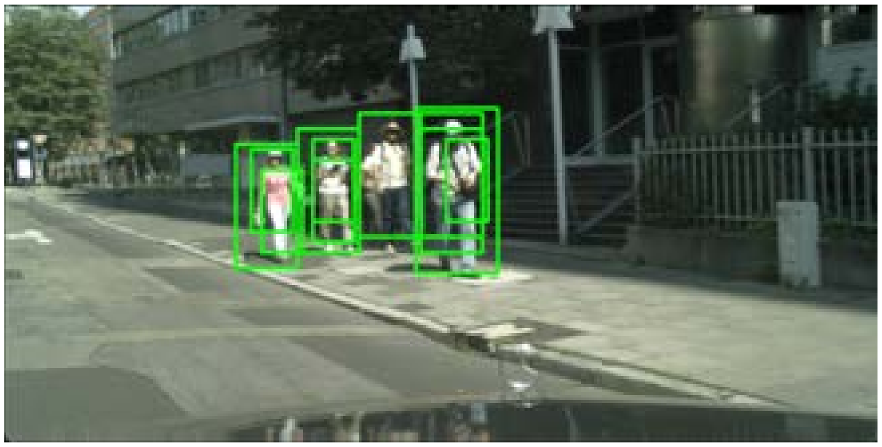

Figure 5.5: Pedestrian detection output before NMS

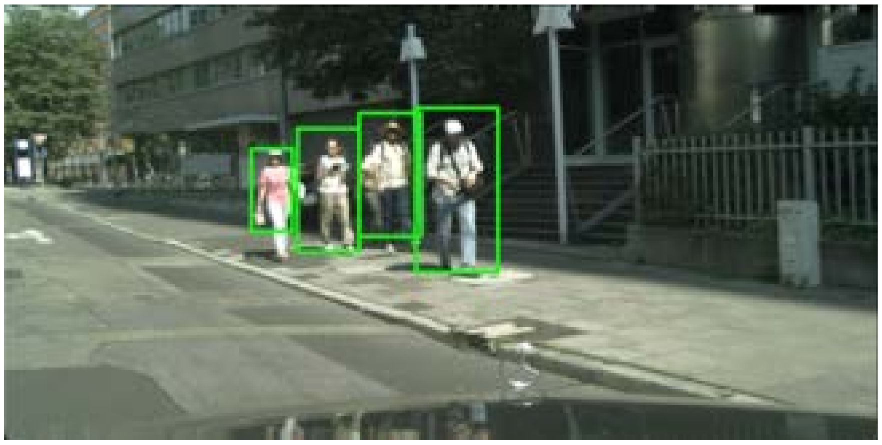

Figure 5.6: Pedestrian detection output after NMS with $T = 0.3$

5.4.3 Color Filtering

One major issue we faced with our original data set was the presence of trees, whose leaves would be consistently confused by pedestrians. In order to avoid this, we can utilize a post-processing method called color filtering.

Definitions

$$H = \{(H, S, V) \mid H_{\text{min}} \leq H \leq H_{\text{max}}, S_{\text{min}} \leq S \leq S_{\text{max}}, V_{\text{min}} \leq V \leq V_{\text{max}}\}$$

$$N = \text{Total number of pixels in the image}$$

$$n = \text{Number of pixels such that the HSV values of the pixel fall within } H$$

$$P_C = \frac{n}{N} \cdot 100$$

Classification

$$\text{Classification} = \begin{cases} \text{Not a pedestrian} & \text{if } P_C \geq T, \\ \text{Potential pedestrian} & \text{if } P_C < T. \end{cases}$$

Results Figure 5.7 highlights the image prior to color filtering, while Figure 5.8 shows our detection results after color filtering. As we can notice, the detection of leaves has drastically reduced after the implementation of color filtering.

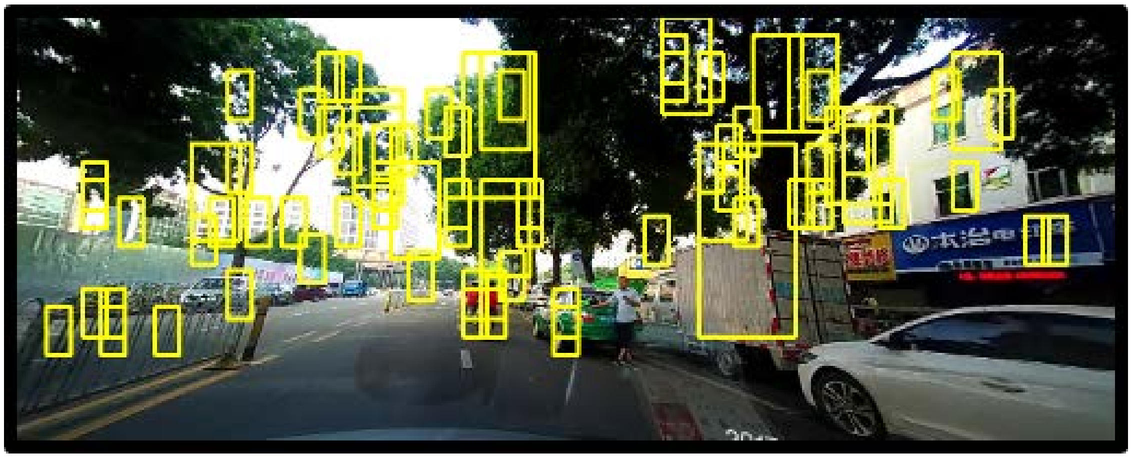

Figure 5.7: Pedestrian detection output before Color Filtering (Thresholded)

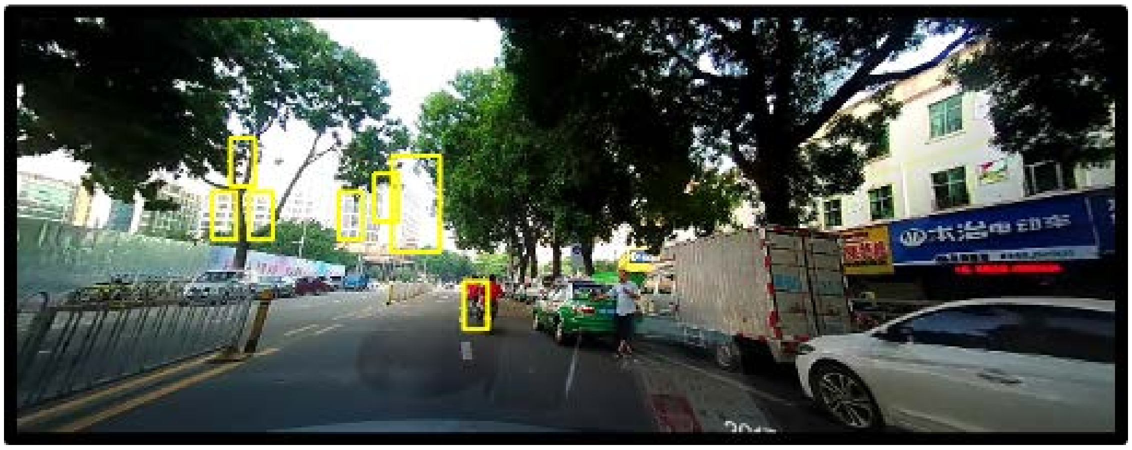

Figure 5.8: Pedestrian detection output after Color Filtering (Thresholded)

6 FINAL RESULTS

6.1 F1 Score

To evaluate the performance of our model with the sliding window, we can use a metric known as the F1 Score. This score is used to specifically measure the accuracy of binary classification models, which is what a support vector machine model falls under [11].

The formula for the F1 Score is the harmonic mean of precision and recall:

$$F1\ score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{6.1}$$

To calculate precision and recall, the number of true positive, false positive, and false negative detections need to be known first. The precision is simply the proportion of true positive detections out of all positive detections, which measures the accuracy of positive detections made by the model. A high precision means that when the model predicts a positive class, it is usually correct, while a low precision means that the model often predicts positives that are incorrect (false positives). It can be calculated via this formula:

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{6.2}$$

The recall is the proportion of true positive detections out of all actual positives (ground truths), which measures the ability of a model to identify all relevant positive instances. A high recall means the model successfully predicts most of the positive instances, while a low recall means that the model misses many actual positives (false negatives). It can be calculated via this formula:

$$recall = \frac{true\ positives}{true\ positives + \ false\ negatives} \tag{6.3}$$

Application When testing our model, we count the number of true positives by seeing if any of our predicted boxes overlap with the ground truths, which can be done by using Equation 5.7. Any that do not overlap are counted as false positives. False negatives are any ground truth boxes that were missed by our predicted boxes.

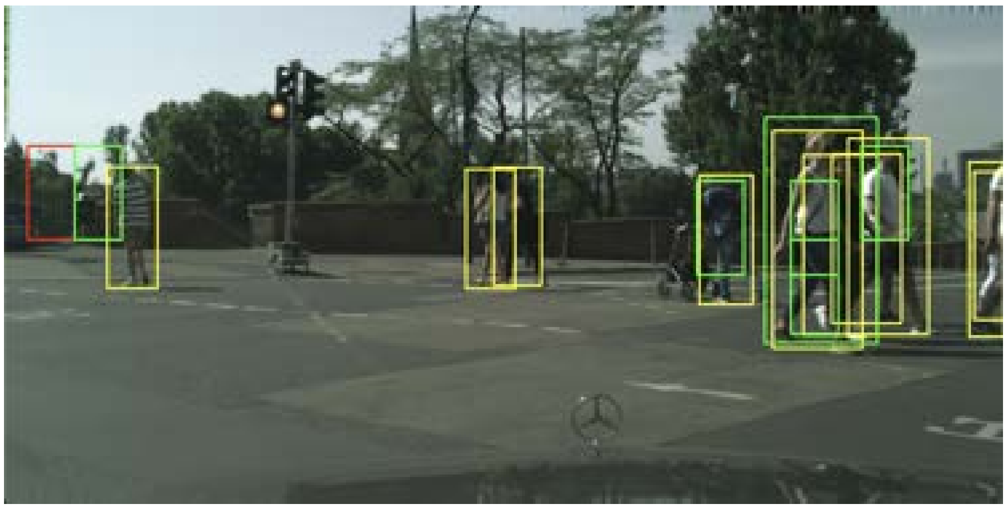

Figure 6.1: Model tested on a validation image. Ground truths are drawn in yellow, true positives are drawn in green, and false positives are drawn in red.

In Figure 6.2, 6 true positives were detected, 1 false positive was detected, and 4 false negatives remained. This resulted in a precision of 0.857, a recall of 0.6, and an F1 Score of 0.706.

The F1 Score can be calculated for not only one image but the entire validation dataset. When running the test on 500 images, we obtained the following graph:

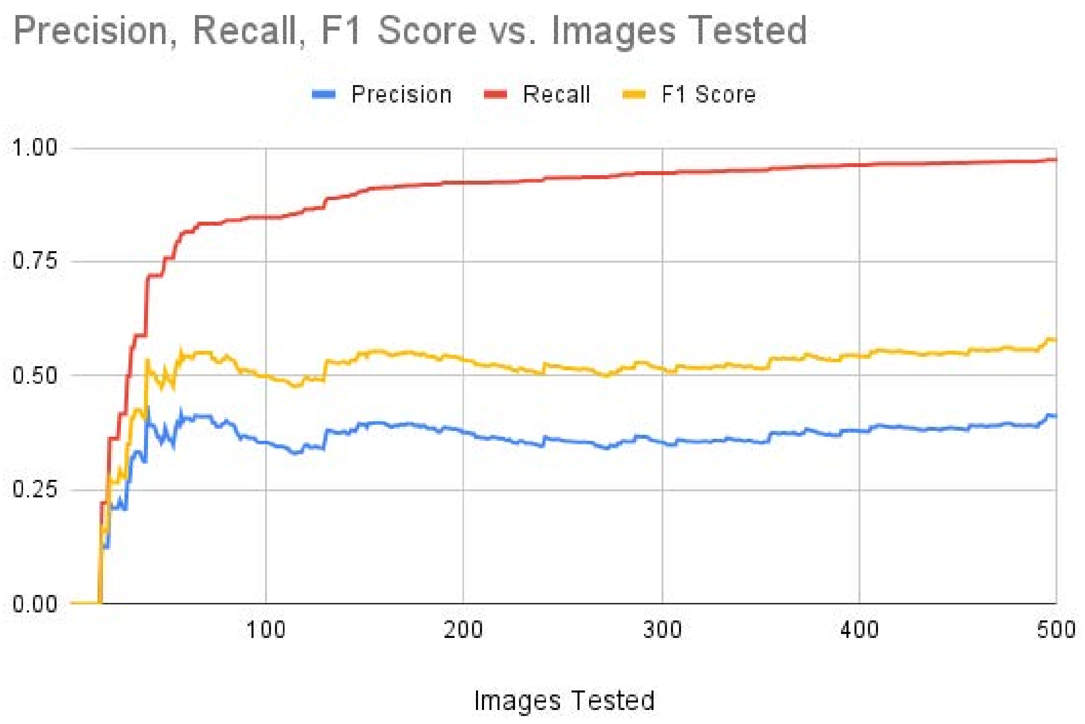

Figure 6.2: Precision, Recall, and F1 Score vs. Number of Images Tested

In the end, our model obtained a precision of 0.412, a recall of 0.973, and an F1 Score of 0.579. This tells us that although the model correctly detects all of the pedestrians in an image most of the time, it is common for it to over-identify, resulting in many false detections/false positives.

6.2 Avoidance Algorithm

In this step, the pedestrian detection algorithm is extended to calculate a direction vector d_u that guides the vehicle to avoid the maximum number of pedestrians while navigating the road safely. The approach involves segmenting the image into regions where no pedestrians are detected and computing the optimal direction based on the widest gap in these regions. The mathematical formulation is as follows:

Image Segmentation and Definitions

Let:

- n be the number of pedestrians detected
- z_i represent segments of the image with no pedestrians
- $L(z_i)$ denote the length of segment z_i in pixels
- $z_{\text{max}} = \max_i L(z_i)$ represents the widest segment with no pedestrians
- $C = \left(\frac{W}{2}, 0\right)$ be the center of the bottom edge of the image, where W is the image width
- $\text{Mid}(z_i)$ denote the midpoint of segment z_i

Direction Vector Calculation The vehicle's direction is determined by the widest pedestrian-free segment y_{max}. Let $\text{Mid}(z_{\text{max}}) = (x_{\text{max}}, y_{\text{max}})$ represent the midpoint of this segment. The direction vector d is computed as:

$$d = (\Delta x, \Delta y) \tag{6.4}$$

where:

$$\Delta x = x_{\text{max}} - C_x \quad \text{and} \quad \Delta y = y_{\text{max}} - C_y$$

The unit direction vector d_u is then given by:

$$d_u = \frac{d}{\|d\|} = \frac{d}{\sqrt{\Delta x^2 + \Delta y^2}} \tag{6.5}$$

Implementation Details Using the pedestrian detection results, the algorithm identifies the widest pedestrian-free region y_{max}. The midpoint of y_{max} serves as the target direction for the vehicle. The computed vector d_u ensures that the vehicle aligns with the safest trajectory to minimize the risk of collision.

Results The direction vector works as intended, guiding the car between the largest gap between pedestrians. As seen in Figure 6.3, the direction vector points right through the gap between pedestrians. However, as seen in Figure 6.4, the direction vector suggests that the car crashes onto the wall. Although this is the largest gap between pedestrians, there is an issue with it. The vector does not account for the depth of the pedestrians, as the pedestrian closest to the car is also closest to the wall. The algorithm still does its job for the most part, however, as we can see from Figure 6.5. A next step could be to weigh the vector towards the sides where pedestrians are farther away.

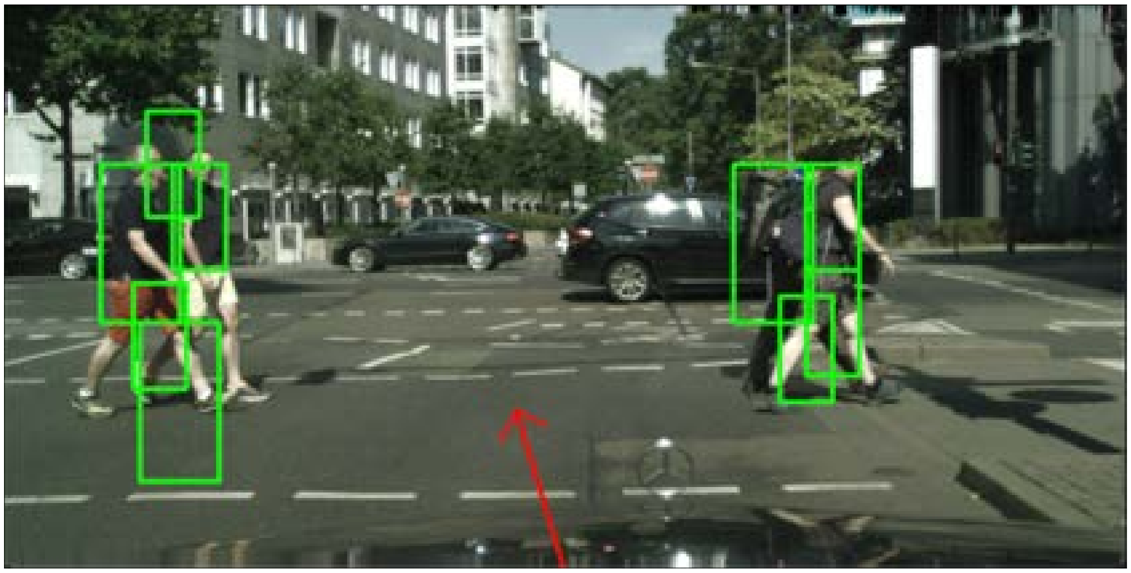

Figure 6.3: $d_u = (-0.29, -0.96)$

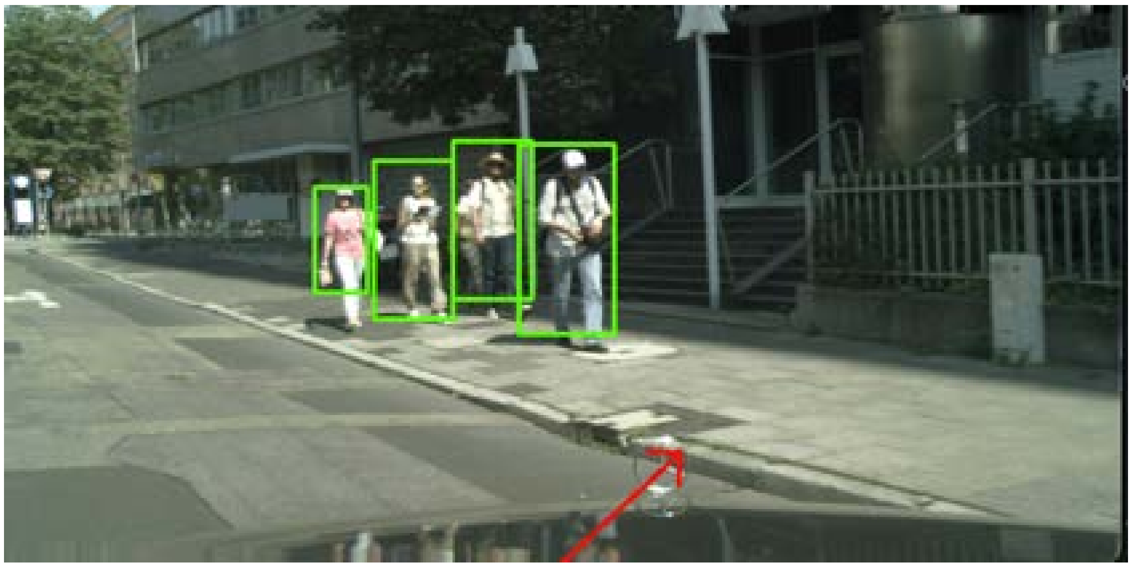

Figure 6.4: $d_u = (0.74, -0.67)$

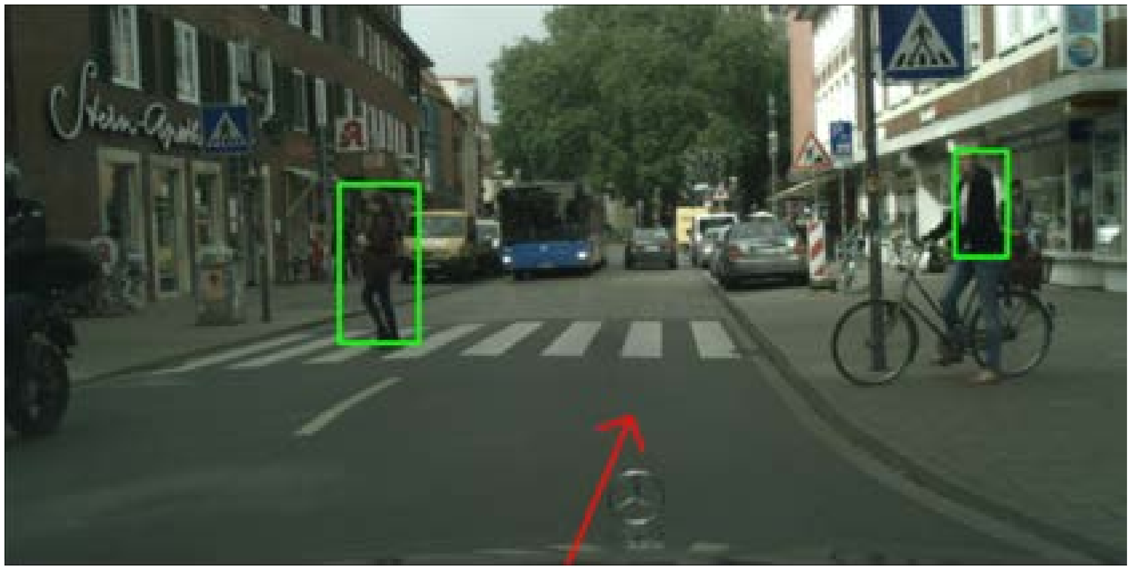

Figure 6.5: $d_u = (0.39, -0.91)$

6.3 Conclusion

In conclusion, we successfully developed and implemented a pedestrian detection system using histogram of oriented gradients for feature extraction, a support vector machine for classification, and a sliding window approach for object detection. The system was trained and evaluated using the Cityscapes dataset, which provided a collection of urban street scenes.

By incorporating techniques such as hard negative mining, threading, thresholding, non-maximum suppression, and color filtering, we improved detection accuracy, reduced redundant bounding boxes, and decreased overall runtime. The performance of our model was evaluated using the F1 Score, which indicated that the model was highly effective at identifying pedestrians, though it tends to generate a significant number of false positive detections.

The final pedestrian detection system is capable of guiding autonomous vehicles in near-crash scenarios to steer in the safest direction, minimizing potential casualties. While the current implementation demonstrates promising results, there remains room for improvement in terms of both computational efficiency and detection accuracy.

6.4 Future Work

The pedestrian detection system developed in this project has demonstrated significant potential, but several areas for improvement and expansion could enhance its performance and applicability. One critical area for future work is the integration of real-time video processing. The current system is designed for static image detection, which limits its deployment in dynamic environments. By implementing dynamic threading or parallel processing techniques to decrease runtime, the system could be optimized to handle continuous video feeds, enabling real-time pedestrian detection.

Further improvements can be made by transitioning from traditional machine learning techniques to deep learning-based approaches. While the combination of histogram of oriented gradients and support vector machines has proven effective, these methods have limitations in complex environments. Integrating deep learning models such as convolutional neural networks, YOLO, or faster R-CNN, could significantly improve detection accuracy and speed, making the system more suitable for real-time applications.

Finally, further refinement of the crash avoidance algorithm is needed. The current algorithm only uses the distance between pedestrians and fails to account for depth in the image. A pedestrian that is closer to the vehicle should receive priority over one that is further away. By using a combination of the size of each bounding box and markings on the road, it may be possible to develop such an algorithm.

7 REFERENCES

[1] W. Ouyang, C. C. Loy, D. Lin, H. Li, Y. Xiong, Q. Huang, D. Zhou, S. Yang, Y. Shen, S. Li, W. Xia, H. Qin, K. Wang, X. Zheng, Q. Li, J. Yan, and Y. Tang, "WIDER face & person challenge 2019 - track 2: Pedestrian detection," 2019. [Online]. Available: https://competitions.codalab.org/competitions/20132

[2] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, 2016. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7780719

[3] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1. IEEE, 2005, pp. 886–893. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1467360&tag=1

[4] M. Cornelison, "Student walking," *UKNow*, 2023. [Online]. Available: https://uknow.uky.edu/sites/default/files/styles/uknow_story_image/public/190129SNOW132%20copy.JPG

[5] H. Wang, J. Xiong, Z. Yao, M. Lin, and J. Ren, "Research survey on support vector machine," in *Proceedings of the 10th EAI International Conference on Mobile Multimedia Communications*. MOBIMEDIA, December 2017. [Online]. Available: https://eudl.eu/pdf/10.4108/eai.13-7-2017.2270596

[6] Larhmam. (2018, October) Svm margin. [Online]. Available: https://commons.wikimedia.org/wiki/File:SVM_margin.png

[7] G. Tanner, "Logistic regression," *Machine Learning Explained*, September 2020. [Online]. Available: https://ml-explained.com/blog/logistic-regression-explained

[8] N. Yang, "How stochastic gradient descent helps find solutions with strong generalization ability in deep learning," *Physical Review Letters*, vol. 130, no. 237101, June 2023.

[9] R. Kleinberg, Y. Li, and Y. Yuan, "An alternative view: When does SGD escape local minima?" in *Proceedings of the 35th International Conference on Machine Learning*, ser. PMLR, vol. 80, 2018, pp. 2698–2707.

[10] S. Fidler. Object detection sliding windows. [Online]. Available: https://www.cs.utoronto.ca/~fidler/slides/CSC420/lecture17.pdf

[11] R. Kundu, "F1 score in machine learning: Intro & calculation," *V7*, December 2022. [Online]. Available: https://www.v7labs.com/blog/f1-score-guide

8 APPENDIX

8.1 Team Member 1: Donny Weintz

Figure 8.1: Donny Weintz

During the project, I helped contribute to the research of pedestrian detection methods, specifically Histograms of Oriented Gradients and Support Vector Machines. I worked to understand the mathematical nuances of these algorithms and how we could utilize them in our pedestrian detector model. In terms of programming, I focused heavily on Histograms of Oriented Gradients and how we could adjust parameters to optimize our feature descriptors. I described our HOG algorithm and parameters in Section 3. I also developed several methods of extracting the best non-pedestrian images, which supplemented the training of the SVM. Finally, I contributed to the development of the sliding window technique and improved its accuracy through various post-processing techniques such as color filtering and threshold.

8.2 Team Member 2: Preetham Yerragudi

Figure 8.2: Preetham Yerragudi

During the project, I was able to utilize my technical and analytical skills to aid our team in many different aspects. I played a crucial part in researching the math behind hard-margin SVM's, such as the optimization problem and Lagrangian equations. I also created a technical implementation of Hard Margin SVM's, utilizing CVXOPT. I was also a significant contributor for the Sliding-Window and the Post-Processing Operations. I wrote implementations for the sliding window algorithm and non-max suppression. I also introduced concurrency into our project, programming threading into both our SVM classifier training and sliding-window operations, improving computational efficiency. I also spearheaded our team's final output, devising the strategy to generate the accident avoidance direction vector and programming its functionality. In addition to all these individual contributions, I also helped our team with the research of HOG, weekly presentations, and the creation of the research poster.

8.3 Team Member 3: Aakarsh N Rai

Figure 8.3: Aakarsh N Rai

I gave the project a solid and efficient foundation on pedestrian detection. My wide knowledge of machine learning and image processing helped me in solving crucial tasks of the project: developing a data augmentation strategy that improved the generalization and robustness of the SVM. I was involved in troubleshooting most parts of the project that included resolving shape mismatches in data augmentation, ensuring proper HOG feature implementation, and refining the heat map to allow for accurate detection. The heat map, designed and created by me helped us visualize regions where the model would give false positives and helped narrow down on certain data augmentation techniques to deal with these issues. Thus, these contributions smoothed all flow in the pipeline with minimal errors, hence highly improving the reliability of the whole system.

8.4 Team Member 4: Aditya Mallepalli

Figure 8.4: Aditya Mallepalli

I contributed to this project by conducting extensive research on soft margin Support Vector Machines (SVMs). This enabled us to fine-tune our classifier for optimal performance in pedestrian detection. I also devised and implemented the half-pedestrian training algorithm, which helped enhance the accuracy of our detection model. Additionally, I implemented the image resizing sliding window technique, significantly improving our computational efficiency while maintaining detection effectiveness. Beyond the technical contributions, I actively collaborated with my team in designing and improving our detection model. I also contributed to the sliding window and abstract sections of the final report. I am sincerely grateful to my teammates for their hard work and our mentors for their guidance throughout this project.

8.5 Team Member 5: Jianing Wang

Figure 8.5: Jianing Wang

I contributed to the visualization of histograms of oriented gradients and the research for the math behind hard margin support vector machines. After finding both the WIDER dataset and the Cityscapes dataset, I used the latter to perform extensive tests for different methods of training our model, which included random negative sampling and hard negative mining. Once our model was trained, I optimized the sliding window by thresholding predictions to cut down on false positives, by tweaking parameters such as window sizes and step size, and by implementing non-maximum suppression to reduce overlapping boxes. To evaluate our model's performance, I ran tests to calculate the F1 Score, precision, and recall using the validation dataset. Additionally, I contributed to this report's abstract and its sections on the dataset, training, and final results.