

# Project Report: Driver Drowsiness Detection System

## Table of Contents

1. Introduction	1
2. System Features and Capabilities	2
3. Technology Stack	3
4. Project Repository Structure	4
5. Data Acquisition and Preprocessing	5
5.1. Dataset Structure Requirements	5
5.2. TensorFlow Data Preprocessing Pipeline	5
5.3. Data Pipeline Visualization	6
6. Model Architecture: Siamese Neural Network	7
6.1. Embedding Network (Shared CNN Backbone)	7
6.2. Siamese Network Structure and Comparison Mechanism	8
6.3. Model Architecture Visualization	8
7. Model Training Configuration	10
8. Current Implementation Status and Identified Issues	10

Document Version: 1.2 Date: 2025-04-26 Abstract: This report details the design, implementation, and current status of a Driver Drowsiness Detection system. The system utilizes computer vision and a Siamese Neural Network architecture to monitor driver alertness by analyzing facial features in real-time, aiming to enhance road safety by mitigating risks associated with driver fatigue. "

## 1. Introduction

Driver fatigue is a critical factor contributing to a substantial percentage of traffic accidents worldwide, often leading to severe injuries or fatalities. The development of robust systems capable of detecting drowsiness in real-time is therefore paramount for improving road safety. This project focuses on the implementation of such a system, employing state-of-the-art computer vision algorithms and a specialized deep learning architecture Ð the Siamese Neural Network.

The core principle involves continuous monitoring of the driver’s facial region using a standard camera. The system extracts and analyzes key physiological indicators strongly correlated with fatigue, including eye closure duration (measured via Eye Aspect Ratio), head pose deviations (nodding, slumping), and mouth movements (specifically yawning). The Siamese network architecture is particularly well-suited for this task as it learns a similarity metric, enabling effective comparison between the driver’s current facial state and baseline 'alert' states, thereby detecting subtle but significant changes indicative of drowsiness onset.

## 2. System Features and Capabilities

The implemented system incorporates the following key functionalities:

- ¥ Real-Time Drowsiness Detection: Provides continuous, frame-by-frame analysis of the driver's face via a camera feed, processing data in real-time to identify emergent signs of fatigue.
- ¥ Eye Aspect Ratio (EAR) Analysis: Leverages facial landmark detection (e.g., using Dlib) to precisely locate the eyes. The EAR is calculated as the ratio of the vertical distance between eye landmarks to the horizontal distance. A significant drop in EAR sustained over a period indicates prolonged eye closure, a strong correlate of drowsiness or microsleep episodes.

*Figure 1. Sample Eye Detection Visualization*

- ¥ Head Pose Estimation (HPE): Analyzes the orientation of the driver's head in 3D space (pitch, yaw, roll). Algorithms estimate these angles from facial landmarks. Abrupt changes, particularly significant downward pitch (nodding) or sustained non-upright poses, are indicative of reduced alertness and potential loss of consciousness. *(Note: The specific HPE algorithm details are abstracted in the current implementation overview but are crucial for robust detection).*
- ¥ Mouth State Analysis and Yawning Detection: Isolates the mouth region using facial landmarks and applies computer vision techniques to detect the characteristic shape deformation and duration associated with yawning, a common physiological response to tiredness.

*Figure 2. Sample Yawn Detection Visualization*

¥ Siamese Neural Network Core: This forms the central intelligence of the system. Instead of classifying a single image into discrete categories (e.g., 'alert', 'drowsy'), the Siamese network takes pairs of images as input (e.g., current frame vs. reference 'alert' frame, or current eye state vs. 'open' eye template). It processes each image through identical Convolutional Neural Network (CNN) branches (with shared weights) to generate high-dimensional feature embeddings. The distance (e.g., L1 or L2) between these embeddings in the feature space quantifies the similarity or dissimilarity between the input images. A final classification layer predicts the probability of the pair belonging to the same state (similar) or different states (dissimilar). This approach is highly effective for detecting subtle deviations from a baseline 'alert' state across multiple facial features.

### 3. Technology Stack

The project leverages a combination of established Python libraries standard within the computer vision and deep learning domains:

- ¥ Python: (Version 3.8+ recommended) Serves as the primary programming language, providing the ecosystem for integrating various libraries.
- ¥ TensorFlow & Keras: (e.g., TensorFlow 2.17.0, Keras 3.6.0) The core deep learning frameworks utilized for designing, building, training, and potentially deploying the Siamese Neural Network model. GPU memory growth configuration is typically employed to prevent out-of-memory errors during intensive training phases.
- ¥ OpenCV (Open Source Computer Vision Library): A fundamental library for a wide range of computer vision tasks, including camera interfacing (video capture), image loading/saving, color space conversions, basic image manipulations (resizing, filtering), and potentially for implementing certain feature extraction steps or classical detection algorithms (like Haar cascades).
- ¥ Dlib: A high-performance C++ library with Python bindings, renowned for its efficient facial landmark detection algorithms (e.g., the 68-point landmark model), which are critical for accurately localizing eyes, mouth, and other facial features required for EAR calculation and

region-of-interest extraction.

- ¥ Imutils: A collection of convenience functions that simplify common image processing tasks built upon OpenCV, such as image rotation, resizing maintaining aspect ratio, translation, and skeletonization, streamlining the development process.
- ¥ Numpy: The cornerstone library for numerical computing in Python, providing efficient multi-dimensional array objects and mathematical operations essential for handling image pixel data and deep learning tensor manipulations.
- ¥ Matplotlib: A comprehensive library for creating static, animated, and interactive visualizations in Python. Used in this project primarily for displaying sample images from datasets, plotting training metrics (loss, accuracy), or visualizing intermediate processing steps during development and debugging.
- ¥ OS module: Python's built-in module for interacting with the operating system, utilized here for tasks like file path manipulation, directory creation/checking, and managing file system interactions required for data loading and model checkpointing.

## 4. Project Repository Structure

The project repository is organized logically to separate configuration, source code, documentation, and pre-trained models (if applicable):

```
.
! " " .editorconfig      # Editor configuration for consistent coding styles
! " " .envrc              # Environment variables configuration (e.g., for
di renv)
! " " .gitattributes      # Defines attributes per path for Git
! " " .gitignore          # Specifies intentionally untracked files for Git
! " " README.adoc        # AsciiDoc version of the main README
! " " README.md           # Original Markdown README
! " " assets/             # Directory for storing sample images or other assets
# ! " " eye.jpg
# $" " yawn.jpg
! " " docs/               # Directory for project documentation and reports
# ! " " data-pipeline.png  # (Generated artifact, not directly linked)
# ! " " model-architecture.png # (Generated artifact, not directly linked)
# ! " " ppt.adoc           # Presentation source file (AsciiDoc)
# ! " " ppt.html           # Rendered HTML presentation
# ! " " project_report.adoc # This report file
# $" " project_report.pdf  # Rendered PDF version of the report
! " " flake.lock          # Nix flake lock file (dependency management)
! " " flake.nix           # Nix flake definition (dependency management)
! " " haarcascade/        # Contains pre-trained Haar cascade classifiers
(alternative/supplementary detection)
# ! " " haarcascade_eye.xml
# $" " haarcascade_frontalface_default.xml
! " " LICENSE             # Project license file (e.g., MIT, Apache 2.0)
! " " pyproject.toml      # Python project metadata and build configuration (PEP
518)
```

```
! " " siamese_network.ipynb    # Jupyter notebook containing the core Siamese network
implementation and experiments
$ " " uv.lock                  # Lock file for 'uv' Python package manager
```

*Note:* A `dataset` directory, crucial for training the model as described in the `siamese_network.ipynb` notebook, is currently absent from the repository structure. This implies that the training data must be sourced or generated externally and placed according to the expected structure.

## 5. Data Acquisition and Preprocessing

Training a Siamese network effectively mandates a carefully curated dataset structured into pairs or triplets of images. This project outlines a data structure based on anchor, positive, and negative samples for learning similarity specific to eye states and yawning.

### 5.1. Dataset Structure Requirements

The `siamese_network.ipynb` notebook anticipates the following directory structure within a top-level `dataset` folder (which needs to be created):

¥ Eyes Dataset: Designed to train the network to distinguish between open and closed eyes relative to an anchor state.

! `dataset/eyes/anchor/`: Contains baseline eye images (e.g., consistently open eyes).

! `dataset/eyes/positive/`: Contains eye images similar to the anchor state (e.g., other open eyes).

! `dataset/eyes/negative/`: Contains eye images dissimilar to the anchor state (e.g., closed eyes).

¥ Yawn Dataset: Designed to train the network to distinguish yawning from non-yawning mouth states.

! `dataset/yawn/anchor/`: Contains baseline mouth images (e.g., closed or neutral mouth).

! `dataset/yawn/positive/`: Contains mouth images similar to the anchor (e.g., other non-yawning mouths).

! `dataset/yawn/negative/`: Contains mouth images dissimilar to the anchor (e.g., various stages of yawning).

*Critical Note:* The successful execution of the data loading and training cells within the provided Jupyter notebook (`siamese_network.ipynb`) is contingent upon the creation of this `dataset` directory and its population with appropriate image files following the specified anchor/positive/negative structure.

### 5.2. TensorFlow Data Preprocessing Pipeline

To manage the complexities of loading, transforming, and batching image data efficiently for deep learning, the notebook implements a `tf.data` pipeline. This pipeline automates the transformation of raw image files into batches suitable for feeding into the Siamese network during training and evaluation. The key stages are:

1. File Path Discovery: `tf.data.Dataset.list_files` is used to locate all relevant image files (e.g., `.jpg`) within the specified anchor, positive, and negative directories. The notebook example uses `.take(300)` potentially as a subset for faster development cycles or due to initial dataset size limitations.
2. Image Loading: `tf.io.read_file` reads the raw byte content of each identified image file.
3. Decoding: `tf.io.decode_jpeg` converts the raw image bytes into TensorFlow tensor format.
4. Resizing: `tf.image.resize` standardizes all images to a fixed spatial dimension (e.g., 105x105 pixels) required by the input layer of the embedding network.
5. Normalization: Pixel intensity values (typically in the range [0, 255]) are scaled to a floating-point range [0, 1] by dividing by 255.0. This normalization step is crucial for stabilizing network training and improving convergence.
6. Pair Formation and Labeling: `tf.data.Dataset.zip` is employed to create the core training pairs. It combines anchor images with corresponding positive images (assigning a label of 1.0 for similarity) and anchor images with corresponding negative images (assigning a label of 0.0 for dissimilarity).
7. Dataset Concatenation: The datasets containing positive pairs and negative pairs are concatenated into a single unified dataset.
8. Caching (`.cache()`): This transformation stores the results of the preceding preprocessing steps (loading, decoding, resizing, normalization) in memory or on local disk after the first epoch. Subsequent epochs can then read directly from the cache, significantly accelerating training by avoiding redundant computations.
9. Shuffling (`.shuffle()`): Randomizes the order of the data samples within a buffer. This is essential for preventing the model from learning spurious correlations based on data order and improves generalization. A sufficiently large `buffer_size` is recommended.
10. Train/Validation/Test Split: The combined dataset is partitioned into distinct sets for training, validation (optional, for hyperparameter tuning), and final testing. The notebook example demonstrates a simple split using `.take()` and `.skip()` for training (70%) and testing (30%).
11. Batching (`.batch()`): Groups individual samples into mini-batches (e.g., size 16). Training is performed iteratively on these batches using stochastic gradient descent (SGD) or its variants (like Adam).
12. Prefetching (`.prefetch()`): Allows the data pipeline to asynchronously prepare subsequent batches of data while the current batch is being processed by the model on the CPU/GPU. This overlaps data preparation and model execution, maximizing hardware utilization and reducing training time.

## 5.3. Data Pipeline Visualization

The following diagram illustrates the flow of data through the described TensorFlow preprocessing pipeline:

# 6. Model Architecture: Siamese Neural Network

The core of the drowsiness detection capability resides in the Siamese Neural Network. This architecture is specifically designed for learning similarity or distance metrics between input pairs, making it ideal for identifying deviations from a baseline 'alert' state.

## 6.1. Embedding Network (Shared CNN Backbone)

The fundamental building block of the Siamese network is a Convolutional Neural Network (CNN) that functions as an embedding generator. This network takes a single input image (preprocessed to 105x105 pixels with 3 color channels) and maps it to a lower-dimensional, dense feature vector  $\mathbb{D}$  the embedding (e.g., 4096 dimensions in the notebook implementation).

Crucially, the exact same embedding network instance, with identical architecture and shared weights, is used to process both images within an input pair (anchor/positive or anchor/negative). Weight sharing ensures that the network learns a consistent mapping function, projecting semantically similar input images to points that are close together in the embedding space, while dissimilar images are mapped farther apart.

The specific CNN architecture detailed in the `siamese_network.ipynb` notebook comprises the following layers:

- ¥ Input Layer: Defines the expected input shape: `(105, 105, 3)`.

- ¥ Convolutional Block 1:

- ! `Conv2D`: 64 filters, 10x10 kernel, ReLU activation. Extracts low-level features like edges and textures. `padding='valid'` implies no zero-padding.

- ! `MaxPooling2D`: 2x2 pool size, `strides=2`. Reduces spatial dimensions (downsampling) and provides a degree of translation invariance.

- ¥ Convolutional Block 2:

- ! `Conv2D`: 128 filters, 7x7 kernel, ReLU activation. Captures more complex patterns by combining features from the previous layer.

- ! `MaxPooling2D`: 2x2 pool size. Further downsampling.

- ¥ Convolutional Block 3:

- ! `Conv2D`: 128 filters, 4x4 kernel, ReLU activation. Continues feature extraction at a coarser spatial resolution.

- ! `MaxPooling2D`: 2x2 pool size.

- ¥ Convolutional Block 4:

- ! `Conv2D`: 256 filters, 4x4 kernel, ReLU activation. Extracts higher-level, more abstract features.

- ¥ Flatten Layer: Reshapes the multi-dimensional feature maps from the final convolutional layer into a single flat vector, preparing it for the fully connected layer.

- ¥ Dense Output Layer: A fully connected layer with 4096 units and a Sigmoid activation function.

This layer produces the final 4096-dimensional embedding vector for the input image. The Sigmoid activation squashes the output values to the range [0, 1].

## 6.2. Siamese Network Structure and Comparison Mechanism

The complete Siamese model integrates two parallel streams using the shared embedding network described above:

1. **Dual Inputs:** The model defines two distinct input layers, one for the `input_image` (e.g., anchor) and one for the `validation_image` (e.g., positive or negative sample).
2. **Parallel Embedding Generation:** Both the `input_image` and the `validation_image` are independently passed through the same shared `embedding` network instance. This yields two embedding vectors, `Emb(A)` and `Emb(B)`.
3. **Distance Calculation:** A custom Keras layer, `L1Dist` (as defined in the notebook), calculates the element-wise absolute difference between the two embedding vectors: `DistanceVector = |Emb(A) - Emb(B)|`. This L1 (Manhattan) distance provides a measure of dissimilarity between the feature representations of the two input images. Other distance metrics like L2 (Euclidean) or cosine similarity could also be used.
4. **Final Classifier:** The calculated `DistanceVector` is fed into a final `Dense` layer. This layer typically has a single output unit with a Sigmoid activation function. It acts as a binary classifier, learning to map the distance vector to a probability score between 0 and 1. This score represents the model's confidence that the original input pair belongs to the same class (i.e., are similar). A score close to 1 indicates high similarity (likely a positive pair), while a score close to 0 indicates low similarity (likely a negative pair).

## 6.3. Model Architecture Visualization

The diagram below illustrates the Siamese network architecture, highlighting the shared embedding network and the comparison process:



## 7. Model Training Configuration

The training process for the Siamese network, as outlined in the notebook, involves configuring the optimizer, loss function, and the training loop itself:

¥ **Optimizer:** The Adam optimizer (`tf.keras.optimizers.Adam`) is selected. Adam is an adaptive learning rate optimization algorithm that is computationally efficient and well-suited for a wide range of deep learning tasks. It maintains per-parameter learning rates that are adapted based on estimates of first and second moments of the gradients. A learning rate of `1e-4` (0.0001) is specified, indicating a relatively small step size for weight updates, often beneficial for stable convergence.

¥ **Loss Function:** `tf.losses.BinaryCrossentropy` is employed. Since the final classifier outputs a probability score between 0 and 1 for the binary task of determining if a pair is similar (label 1.0) or dissimilar (label 0.0), binary cross-entropy is the appropriate loss function. It measures the difference between the predicted probability and the true binary label.

¥ **Custom Training Loop:** Instead of relying solely on Keras' `model.fit()`, the notebook defines a custom training loop using `@tf.function`. Decorating the `train_step` function with `@tf.function` compiles it into a TensorFlow graph, which can lead to significant performance improvements by optimizing operations and reducing Python overhead, especially when executing on GPUs or TPUs.

! **`train_step` Function:** This function encapsulates the operations for a single training batch. It performs a forward pass through the Siamese model (`siam`) to get predictions, calculates the loss using the `BinaryCrossentropy` function against the true labels (`y`), computes gradients of the loss with respect to the model's trainable variables using `tf.GradientTape`, and finally applies these gradients to update the model weights using the configured `optimizer.apply_gradients()`.

! **`train` Function:** This outer function orchestrates the training over multiple epochs. It iterates through the specified number of `EPOCHS`, processing the training dataset batch by batch using the `train_step` function. It includes progress reporting using `tf.keras.utils.Progbar` to provide visual feedback on training progress within each epoch.

¥ **Checkpointing:** `tf.train.Checkpoint` is utilized for saving the model's state periodically during training. This includes saving the weights of the Siamese model (`siam`) and the state of the optimizer (`opt`). Checkpoints are configured to be saved every 4 epochs to the `./training_checkpoints` directory. This allows training to be interrupted and resumed later without losing progress, and also provides backups of the model at different stages.

## 8. Current Implementation Status and Identified Issues

While the `siamese_network.ipynb` notebook provides a comprehensive architectural blueprint and training setup, analysis of the notebook's execution state (assuming it reflects recent runs) reveals critical issues preventing successful execution:

1. **`TypeError` within `L1Dist` Layer:** A `TypeError: unsupported operand type(s) for -: 'list' and`

'list' is reported during the instantiation or calling of the custom `L1Dist` layer within the `make_siamese()` function. This strongly suggests that the inputs being passed to the layer's `call()` method (`input_embedding` and `validation_embedding`) are Python lists rather than the expected TensorFlow tensors. This likely originates from an error in how the outputs from the shared `embedding` network are captured and passed when defining the functional Keras model for the Siamese structure. The subtraction operation (`-`) is not defined for standard Python lists in this context. This structural error in the model definition must be resolved before the model can be successfully built or trained.

2. `NameError` for `siam` Variable: As a direct consequence of the `TypeError` preventing the successful execution of `make_siamese()`, the variable `siam`, intended to hold the compiled Siamese Keras model, is never assigned. Subsequent code cells attempting to use this variable, such as the `tf.train.Checkpoint` setup (`ckpt = tf.train.Checkpoint(opt=opt, siamese_model=siam)`), fail with a `NameError: name 'siam' is not defined`.
3. Missing Training/Testing Dataset: The data loading and preprocessing sections explicitly rely on the existence of a `dataset` directory structured with `eyes` and `yawn` subdirectories, further containing `anchor`, `positive`, and `negative` image folders. As noted in the Project Structure section, this `dataset` directory is absent from the repository's file listing. The system cannot be trained or evaluated without acquiring or generating the necessary image data and organizing it according to this predefined structure.

**Conclusion on Status:** The project provides a solid conceptual foundation and code structure for a Siamese network-based drowsiness detector. However, it is currently non-operational due to fundamental errors in the model's layer connection logic (`TypeError` in `L1Dist`) and the critical absence of the required training dataset. Priority next steps involve debugging the Keras model definition to ensure correct tensor flow between the embedding network and the distance layer, and subsequently acquiring and structuring the necessary image datasets for training and evaluation.