



Final Report

Yassin Abdelaal 120210002

Moustafa Rezk 120210004

Shahd Zaki 120210017

Submitted to:

Dr.Ahmed Gomaa

Abstract

Driver distraction is a major cause of road accidents, and detecting distractions in real time can help enhance driver safety. In this project, we developed a system to classify driver behavior into distracted and non-distracted categories using image data. Not only that but classify what kind of distraction the driver is doing like texting and talking on the phone and also classifies what hand they are using and also classifies if they are reaching for something behind them or talking to the passenger or operating the radio or fixing their or applying makeup.

Teaser Figure





c6



c7



c8



c9



c6



c7



c8



c9



Introduction

Driver distraction significantly contributes to road accidents, with causes ranging from mobile phone usage to interacting with passengers. Detecting such distractions in real time can enable interventions that may prevent collisions. The project falls under computer vision and deep learning, specifically focused on behavioral classification from RGB images captured inside vehicles to 10 different classes divided to one safe driving class and 9 other which fall under distracted driving.

The dataset used was Kaggle's State Driver Distraction Dataset and 3 methods were incorporated a custom CNN model and finally EfficientNetB3.

Approach

Dataset

In our project, we used the State Farm distraction-detection dataset. The dataset was made available on Kaggle in 2016 as part of a contest. It has been used in numerous

research, to identify driver distraction. Ten classes are present in the State Farm dataset, which are as follows:

Classes	Action taken	Number of Images
C0	Safe driving	2489
C1	Texting - right	2267
C2	Talking on the phone - right	2317
C3	Texting – left	2346
C4	Talking on the phone - left	2326
C5	Operating the radio	2312
C6	Drinking	2325
C7	Reaching behind	2002
C8	Hair and makeup	1911
C9	Talking to a passenger	2129
Total		22424

Data Preprocessing

In this project, we utilize a deep learning approach to detect driver distraction using images. A critical part of this process involves preparing and augmenting the dataset to enhance the model's performance and robustness. The dataset is split into training and validation sets, and data augmentation techniques are applied to the training set to create a more diverse set of images. This report outlines the methodology used for data preparation and augmentation.

Data Augmentation

Data augmentation is a technique used to increase the diversity of the training dataset without actually having to collect new data. This is achieved by applying various transformations to the already existing images in our dataset, such as shifting, rotating,

and zooming. These transformations help the model generalize better by exposing it to a variety of image conditions during training.

These augmentations help in simulating real-world scenarios where the driver's environment might change due to slight movements, different angles, or varying distances.

Data Splitting

The dataset is split into two subsets: training and validation. The training set is used to train the model, while the validation set is used to evaluate the model's performance and ensure it is not overfitting. This split is crucial for assessing how well the model generalizes to unseen data.

The dataset is organized into directories, with separate folders for each class of images. Then the data is split like this:

Training Set: This subset includes 80% of the data and is used for model training. Augmentation is applied to this set to create more diverse training samples.

Validation Set: This subset includes 20% of the data and is used to validate the model during training. No augmentation is applied to the validation set to ensure a fair evaluation.

CNN model

Before we get to our own CNN model we are going to explain what is CNN model and its layers,

A Convolutional Neural Network (CNN) is a class of deep neural networks that were specifically designed for tasks involving images and other structured grid data. CNNs have revolutionized the field of computer vision and are widely used for tasks such as image classification, object detection, segmentation, and even more. They are inspired by the organization of the animal visual cortex, where neurons respond to overlapping regions of the visual field, allowing a hierarchical representation of visual features.

1. **Input Layer:** The input layer of a CNN accepts the raw input data, typically in the form of images. Images are represented as grids of pixel values, where each pixel corresponds to a specific color intensity in the image. In grayscale images, each pixel value represents the brightness of that pixel, while in color images, each pixel has three values corresponding to the intensities of the red, green, and blue (RGB) color channels.

2. **Convolutional Layers:** Convolutional layers are the core building blocks of CNNs. They consist of a set of learnable filters (also called kernels) that are convolved with the input data to extract features. Each filter detects specific patterns or features, such as edges, textures, or shapes, at different spatial locations in the input data. During training, the CNN

learns the optimal values for these filters to extract relevant features from the input. Convolutional layers preserve the spatial relationships between pixels in the input data, allowing the network to learn hierarchical representations of features.

3. Activation Function: An activation function is applied element-wise to the output of each convolutional layer. Common activation functions include Rectified Linear Unit (ReLU), sigmoid, and hyperbolic tangent (tanh). ReLU is the most widely used activation function in CNNs due to its simplicity and efficiency. It introduces non-linearity to the model, enabling it to learn complex mappings between inputs and outputs.

4. Pooling Layers: Pooling layers are used to reduce the spatial dimensions of feature maps produced by convolutional layers while retaining the most important information. The most common type of pooling is max pooling, where the maximum value within each region of the feature map is retained, effectively downsampling the spatial dimensions. Pooling layers help to make the representations learned by the CNN more invariant to small translations and distortions in the input data, while also reducing the computational complexity of the network.

5. Fully Connected Layers: Fully connected layers, also known as dense layers, are typically added at the end of the CNN architecture to perform high-level reasoning and classification. And transform our matrix of features into a 1 column matrix. Each neuron in a fully connected layer is connected to every neuron in the previous layer, allowing the network to learn complex relationships between features.

The output of the fully connected layers is often passed through a softmax activation function to produce class probabilities for multi-class classification tasks.

6. Dropout Layers: Dropout layers are a form of regularization used to prevent overfitting in CNNs. During training, a fraction of neurons in the dropout layer are randomly "dropped out" (set to zero) with a specified probability. Dropout helps to prevent the network from relying too heavily on any particular set of features, forcing it to learn more robust and generalizable representations.

Batch Normalization Layers: Batch normalization layers are used to normalize the activations of each layer across mini-batches of data during training.

Our CNN Model

The model is designed to classify images, making it suitable for tasks such as recognizing objects in photographs or detecting features in medical images. The architecture includes multiple convolutional layers for feature extraction, followed by dense layers for

classification. Regularization techniques such as dropout and batch normalization are incorporated to prevent overfitting and ensure stable training.

Input Layer:

The input layer is defined to accept images of a specific size (`img_rows` x `img_cols`) and color depth (`color_type`). This layer serves as the entry point of the model, receiving the pixel data from the images.

Convolutional Layers:

The model we created includes four convolutional layers, each followed by an activation function which is called LeakyReLU activation which is pretty similar to the Famous ReLU but solves problems a common Problem with ReLU which is known as the Dying ReLU problem, batch normalization, max pooling, and dropout. These layers work as follows:

Convolutional Layer: Applies 2D convolution operations to the input, using filters to detect various features like edges and textures. Each convolutional layer in the model increases the depth of the feature maps, starting from 32 filters in the first layer to 256 in the last.

LeakyReLU Activation: Introduces non-linearity to the model, allowing it to learn more complex patterns. The LeakyReLU activation is chosen for its ability to avoid the problems with ReLU which is the dying ReLU problem which are the vanishing gradient problem by allowing a small gradient when the input is negative, and dying Neurons during back propagation.

Batch Normalization: Normalizes the activations of the previous layer at each batch, which speeds up training and provides some regularization, reducing the need for other forms of regularization.

Max Pooling: Reduces the spatial dimensions (height and width) of the feature maps, retaining the most important features while reducing computational load.

Dropout: Randomly sets a fraction of the input units to zero at each update during training time, which helps prevent overfitting by making the model more robust.

Global Average Pooling: After the last convolutional block, a Global Average Pooling layer is applied. This layer computes the average of each feature map, converting the 2D feature maps into a single long feature vector per image. This reduces the number of parameters and helps prevent overfitting while maintaining spatial information.

Dense Layers: Following the Global Average Pooling, the model has three dense (fully connected) layers with 512, 256, and 128 units respectively. Each dense layer is followed by:

LeakyReLU Activation: We use it similarly to the convolutional layers, to introduce non-linearity and avoid making a biased linear model.

Batch Normalization: Normalizes the outputs of the dense layers to stabilize and speed up training.

Dropout: Applied with a higher dropout rate (0.5) to further prevent overfitting, especially since these layers have a high number of parameters.

Output Layer: The final layer is a dense layer with a number of units equal to the number of classes (`num_classes`). It uses a softmax activation function, which outputs a probability distribution over the classes, allowing the model to make a prediction about the class of the input image.

Model Summary:

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 64, 64, 3)	0
conv2d (Conv2D)	(None, 64, 64, 32)	896
leaky_re_lu (LeakyReLU)	(None, 64, 64, 32)	0
batch_normalization (BatchNormalization)	(None, 64, 64, 32)	128
max_pooling2d (MaxPooling2D)	(None, 32, 32, 32)	0
dropout (Dropout)	(None, 32, 32, 32)	0
conv2d_1 (Conv2D)	(None, 32, 32, 64)	18,496
leaky_re_lu_1 (LeakyReLU)	(None, 32, 32, 64)	0
batch_normalization_1 (BatchNormalization)	(None, 32, 32, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 64)	0
dropout_1 (Dropout)	(None, 16, 16, 64)	0
conv2d_2 (Conv2D)	(None, 16, 16, 128)	73,856
leaky_re_lu_2 (LeakyReLU)	(None, 16, 16, 128)	0
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 8, 8, 128)	0
dropout_2 (Dropout)	(None, 8, 8, 128)	0
conv2d_3 (Conv2D)	(None, 8, 8, 256)	295,168
leaky_re_lu_3 (LeakyReLU)	(None, 8, 8, 256)	0
batch_normalization_3 (BatchNormalization)	(None, 8, 8, 256)	1,024
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 256)	0
dropout_3 (Dropout)	(None, 4, 4, 256)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 256)	0
dense (Dense)	(None, 512)	131,584

leaky_re_lu_4 (LeakyReLU)	(None, 512)	0
batch_normalization_4 (BatchNormalization)	(None, 512)	2,048
dropout_4 (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 256)	131,328
leaky_re_lu_5 (LeakyReLU)	(None, 256)	0
batch_normalization_5 (BatchNormalization)	(None, 256)	1,024
dropout_5 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 128)	32,896
leaky_re_lu_6 (LeakyReLU)	(None, 128)	0
batch_normalization_6 (BatchNormalization)	(None, 128)	512
dropout_6 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1,290

In our approach to driver distraction detection, we utilized EfficientNetB3 as the backbone of our image classification model. EfficientNetB3, available through `tensorflow.keras.applications`, was chosen for its efficient scaling capabilities and strong performance on a wide range of image recognition tasks. Although the original EfficientNetB3 architecture is designed for 300x300 input resolution, we adapted it for 224x224x3 input dimensions to balance performance with reduced computational overhead, making the model more suitable for real-time applications and resource-constrained environments.

We initialized the model with pretrained ImageNet weights, setting `include_top=False` to remove the original classification head. This allowed us to add a custom classification head tailored to our 10-class driver distraction data set. Specifically, we added the following layers:

- A Global Average Pooling 2D layer to reduce the spatial dimensions of the output feature maps.
- A Dense layer with 512 units and ReLU activation to introduce non-linearity and learn task-specific features.
- A final Dense layer with 10 units and softmax activation to output class probabilities corresponding to the 10 distraction categories.

To preserve the high-level feature representations learned during pretraining, we froze all layers of the base EfficientNetB3 model, training only the newly added top layers. This transfer learning strategy is effective when the available dataset is relatively small or domain-specific.

The model was compiled using the RMSprop optimizer, known for handling sparse gradients well, and the categorical cross-entropy loss function, appropriate for multi-class classification tasks. We tracked accuracy as the primary evaluation metric.

Training was conducted over 20 epochs using the prepared training and validation datasets. We monitored validation performance to detect potential overfitting. While RMSprop was used initially, in later stages of training (not shown in the code snippet), the model could optionally be fine-tuned by unfreezing select deeper layers of the base model and retraining with a lower learning rate.

This EfficientNetB3-based approach allowed us to leverage the power of a state-of-the-art convolutional architecture while maintaining a manageable model size and ensuring generalization through transfer learning

Experiments and results

Both Models tested CNN and EfficientNetB3 had the same data splitting and augmentation and ResNet50 had a bit more.

Augmentation

Width Shift: Images are randomly shifted horizontally by a certain fraction of the total width.

Height Shift: Images are randomly shifted vertically by a certain fraction of the total height.

Shear: Images are distorted along an axis, altering the angles of the content.

Zoom: Images are randomly zoomed in or out.

Rotation: Images are randomly rotated.

Horizontal flip: Images are randomly flipped.

Splitting


Training Set: This subset includes 80% of the data and is used for model training.

Augmentation is applied to this set to create more diverse training samples.

Validation Set: This subset includes 20% of the data and is used to validate the model during training. No augmentation is applied to the validation set to ensure a fair evaluation.

Only difference is the input shape in CNN is 64 and EfficientNetB3 is 224

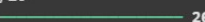
CNN model

Epoch 30/30
449/449  78s 171ms/step - accuracy: 0.8006 - loss: 1.1824 - val_accuracy: 0.7489 - val_loss: 1.3921

EfficientNetB3

Epoch 20/20
449/449  261s 575ms/step - accuracy: 0.9075 - loss: 0.2842 - val_accuracy: 0.9402 - val_loss: 0.1970

ResNet50

Epoch 20/20
449/449  265s 591ms/step - accuracy: 0.9861 - loss: 0.0451 - val_accuracy: 0.9877 - val_loss: 0.0432

Evaluation

All models were evaluated through training and validation accuracy and loss

Qualitative Results

We used the test data available in the state farm dataset which is isolated from train and test data to test our ResNet50 and EfficientNet50 which showed best results.

Correct

Random Image: img_83341.jpg



1/1 ————— 7s 7s/step
Predicted Class: Reaching behind

Random Image: img_94580.jpg



1/1 ————— 7s 7s/step
Predicted Class: Reaching behind

Random Image: img_9081.jpg



1/1 ————— 6s 6s/step
Predicted Class: Operating the radio

Random Image: img_42390.jpg



1/1 ————— 6s 6s/step
Predicted Class: Talking on the phone - left

Random Image: img_99398.jpg



1/1 ————— 7s 7s/step
Predicted Class: Reaching behind

Random Image: img_2984.jpg



1/1 ————— 7s 7s/step
Predicted Class: Talking on the phone - right

Random Image: img_97363.jpg




1/1 ————— 6s 6s/step
Predicted Class: Texting - right

Incorrect


Random Image: img_53238.jpg



1/1  6s 6s/step
Predicted Class: Hair and makeup

Random Image: img_4167.jpg



1/1  7s 7s/step
Predicted Class: Texting - left

In conclusion, we were able to try 2 methods to train a successful model the convolutional neural network that we made and discussed which showed lower accuracy but worse predictions were most outputs were wrong while the EfficientNetB3 showed higher accuracy and had a high correct accuracy on the test data.

To transition our project from a research prototype to a practical real-world application, several enhancements, and additional steps are required:

Increase the accuracy: Try to increase the accuracy on our model for real-life scenarios where accuracy is important and make a unbiased model.

Video integration and Edge Device Integration: Deploy the model on edge devices such as Raspberry Pi, NVIDIA Jetson, or smartphones. For example: Imagine installing a small, powerful computer like a Raspberry Pi in your car. This device runs your trained model and monitors the driver's actions using a camera. Develop a mobile application that uses the phone's camera to monitor the driver and detect distracted behavior, providing real-time feedback and alerts.

Driver Alerts: Create a user-friendly interface that provides real-time alerts to the driver via visual, auditory, or haptic feedback. Example: If the system detects that the driver is texting, it could sound a beep (audio alert) and display a warning message on the car's dashboard screen (visual alert). Integrate the system into the car's dashboard for seamless interaction. Work with car manufacturers to integrate this alert system into the car's dashboard. This way, alerts are shown directly on the car's built-in screen, ensuring they are noticeable without adding extra distractions.

References

**[1] *Convolutional Neural Network (CNN) : Tensorflow Core*. TensorFlow. (n.d.).
<https://www.tensorflow.org/tutorials/images/cnn>**

**[2] GeeksforGeeks. (2024, March 14). *Introduction to convolution neural network*.
<https://www.geeksforgeeks.org/introduction-convolution-neural-network/>**

**[5] Detection of distracted driver using convolution neural ... (n.d.).
<https://arxiv.org/pdf/2204.03371>**

**[6] Automatic driver distraction detection using deep convolutional neural networks
<https://www.sciencedirect.com/science/article/pii/S2667305322000163>**