



SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

50.035 Computer Vision 2023
Final Project Report
Competition Track
State Farm Distracted Driver Detection

Group 8 Members

1005802 - Swastik Majumdar
1005265 - Win Tun Kyaw
1005284 - Ishan Monnappa Kodira
1005375 - Aditi Kumaresan
1005374 - Nguyen Thai Huy
1005372 - Prabhakar Dhilahesh

Table of Contents

Executive Summary	2
Problem Statement	3
Kaggle Dataset	4
Data Augmentation	6
Models	8
1. Custom CNN	8
2. Transfer learning with DenseNet	9
3. VGG - 16 model	11
4. ResNet-50	12
5. EfficientNet	13
6. Ensemble Learning	16
Results & Evaluation	17
Kaggle	17
Testing	19
Future Development	21
Individual Contributions	22
References	23
Appendix	24

Executive Summary

In response to the escalating concerns around distracted driving, our computer vision project sought to deploy simple yet effective solutions for the detection of distracted behaviors using dashboard camera images. With this in mind, our group has decided to join a State Farm Distracted Driver Detection Kaggle challenge to devise a model capable of categorizing driver behaviors from a dataset consisting of $\approx 79,700$ training images and $\approx 22,400$ testing images with precision.

This report outlines the various solutions created to solve the Kaggle challenge. Upon utilization of data augmentation on the train dataset, our solution applied computer vision concepts and algorithms by experimenting with a multitude of approaches to seek what suited this project best, namely: Custom CNN, DenseNet, VGG-16, ResNet, EfficientNet, and an Ensemble Learning of all previous models.

Significant strides were made in model performance through Ensemble Learning, demonstrating its prowess as the best-performing model with the lowest cross-entropy loss and highest F1 score. However, upon reflection for potential improvements, further refinements were identified. These include the implementation of region of interest (ROI) pooling to focus on critical regions of interest and the adoption of advanced techniques like fine-tuning, batch normalization, enhanced image augmentations, dropout layers, hyperparameter tuning, and a combination of these techniques to further mitigate overfitting and enhance model robustness. The code of the project can be found in the [Appendix](#) section.

Problem Statement

According to the Centers for Disease Control and Prevention's (CDC) Motor Vehicle Safety Division, one in five car accidents is caused by a distracted driver. The ramifications are stark, leading to an annual toll of 425,000 injuries and 3,000 fatalities. Road safety problems are a major concern and human behavior is ascribed as one of the main causes and accelerators of the problems. Driver distraction has been identified as the main reason for accidents. Distractions can be caused due to reasons such as mobile usage, drinking, operating instruments, applying makeup, and social interaction.

State Farm aims to explore the potential of leveraging computer vision to automatically identify distracted driving behaviors and has provided a dataset of dashboard camera images, challenging participants on Kaggle to classify behaviors in each image. The task encompasses identifying whether a driver is operating their vehicle attentively, or engaging in distracting activities such as mobile usage, drinking, and operating instruments.

For the scope of this project, we will focus on leveraging highly efficient pretrained models to classify different driver distractions at runtime using computer vision. We would also analyze and compare the overall accuracy of each model.

Kaggle Dataset

The dataset we used for the project is sourced from Kaggle, which comprises a total of $\approx 79,700$ labeled images taken from dashboard cameras for training purposes, alongside an additional $\approx 22,400$ images for testing. To further characterize the dataset, the train and test data are split on the drivers, such that one driver can only appear on either train or test set.



Figure 1. Samples of labeled images from the train dataset

From Kaggle, we were also provided with the ground truth labels for each image, categorizing the driver's behavior into one of the following ten classes:

- c0: Safe Driving
- c1: Texting - Right
- c2: Talking on the Phone - Right
- c3: Texting - Left
- c4: Talking on the Phone - Left
- c5: Operating the Radio
- c6: Drinking
- c7: Reaching Behind
- c8: Hair and Makeup
- c9: Talking to Passenger

The data is also well distributed, as visualized in the following figure.

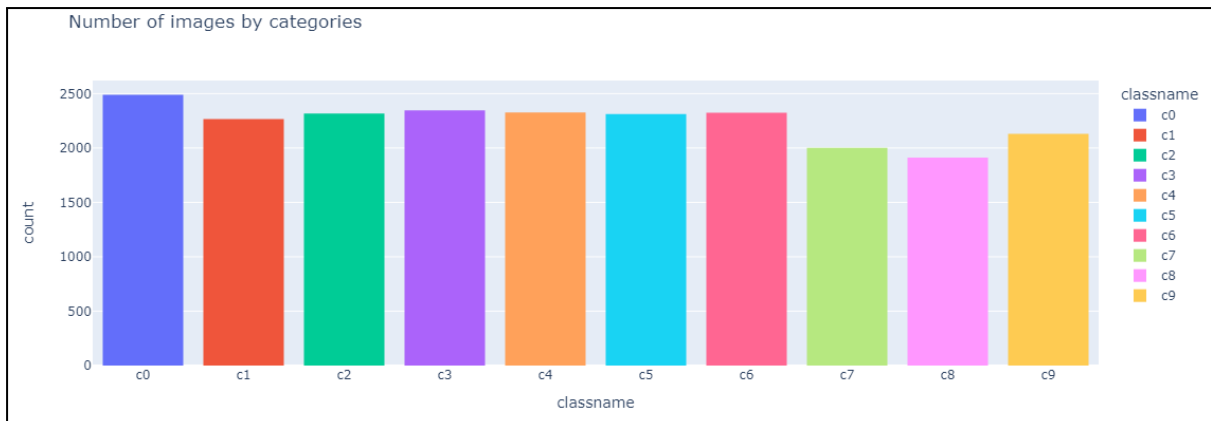


Figure 2. Bar chart of training images classified by their ground truth labels, plotted using Plotly.

It should also be noted that State Farm conducted experiments in a controlled environment, utilizing a truck to tow the car around on the streets. Thus, the "drivers" in the images were not actually driving, providing context for the experimental conditions.

Data Augmentation

In computer vision problems, image augmentation plays a pivotal role in enhancing the diversity of the training set and, consequently, the robustness of the trained models. As the Kaggle-provided dataset is inherently visual and lacks the structured features of tabular data, image augmentation becomes a crucial step in the pre-processing pipeline. Therefore, to introduce variability into the training dataset, we leverage the Augmentor library, creating a customized augmentation pipeline.

```
def augment_images_in_folder(input_folder, output_folder, num_of_augment_img):
    # Create an Augmentor pipeline for the current image
    pipeline = Augmentor.Pipeline(input_folder, output_directory=output_folder)

    # Add augmentations to the pipeline
    pipeline.flip_left_right(probability=0.8)
    pipeline.shear(probability=0.5, max_shear_left=15, max_shear_right=15)
    pipeline.rotate(probability=0.5, max_left_rotation=20, max_right_rotation=20)

    pipeline.random_brightness(probability=0.2, min_factor=0.5, max_factor=1.5)
    pipeline.random_contrast(probability=0.2, min_factor=0.5, max_factor=1.5)
    pipeline.random_color(probability=0.2, min_factor=0.5, max_factor=1.5)

    pipeline.skew(probability=0.5)
    pipeline.histogram_equalisation(probability=0.5)

    pipeline.sample(num_of_augment_img)
    pipeline.augmentor_images.clear()

def unzip_and_augment(zipped_file_path, num_of_augment_img):
    if not os.path.isfile(zipped_file_path):
        file_path = zipped_file_path
    else:
        file_path = unzip_file(zipped_file_path)

    train_folder_path = os.path.join(file_path, "train")

    train_folder_path = os.path.join(file_path, "train")
    for subfolder_name in os.listdir(train_folder_path):
        subfolder_path = os.path.join(train_folder_path, subfolder_name)
        # Check if the current item in the directory is a directory itself
        if os.path.isdir(subfolder_path):
            # Create an output path for the augmented images
            output_path = os.path.join('..', subfolder_name)

            # Call the function to augment images in the current subfolder
            augment_images_in_folder(subfolder_path, output_path, num_of_augment_img)

    file_path = os.path.join('imgs')

    num_of_augment_img = 250

    unzip_and_augment(file_path, num_of_augment_img)
```

Initialised with 2489 image(s) found.
Output directory set to imgs\train\c0\..\c0.
Processing <PIL.Image.Image image mode=RGB size=640x480 at 0x27238D1A700>: 100% | 250/250 [00:01<00:00, 180.45 Samples/s]
Initialised with 2267 image(s) found.
Output directory set to imgs\train\c1\..\c1.
Processing <PIL.Image.Image image mode=RGB size=640x480 at 0x27238D3B370>: 100% | 250/250 [00:01<00:00, 166.36 Samples/s]
Initialised with 2317 image(s) found.
Output directory set to imgs\train\c2\..\c2.
Processing <PIL.Image.Image image mode=RGB size=640x480 at 0x2723876A040>: 100% | 250/250 [00:01<00:00, 170.23 Samples/s]
Initialised with 2346 image(s) found.

Figure 3. Codes to augment the train dataset

The augmentation techniques employed include horizontal flipping, shearing, rotation, random adjustments to brightness, contrast, and color, skewing, and histogram equalization. These techniques collectively introduce variations in lighting, orientation, and spatial characteristics, enriching the dataset with diverse instances of distracted driving scenarios (Madhugiri, 2021). To every class in the training dataset, 250 augmented images were added, expanding the dataset by 2500.

In addition to augmentation, we also split our data into two parts: one designated for **RGB** training and the other for **Grayscale**. This bifurcation allows our models to learn from both color and grayscale representations, catering to potential variations in lighting conditions and enhancing the adaptability of the trained models.

```
grayscale3_train = train_generator_grayscale.flow(x_train_gray3, y_train, batch_size=batch_size, subset='training')
grayscale3_val = train_generator_grayscale.flow(x_train_gray3, y_train, batch_size=batch_size, subset='validation')
grayscale3_test = test_generator_grayscale.flow(x_test_gray3, y_test, batch_size=batch_size)

rgb_train = train_generator_color.flow(x_trainRGB, y_trainRGB, batch_size=batch_size, subset='training')
rgb_val = train_generator_color.flow(x_trainRGB, y_trainRGB, batch_size=batch_size, subset='validation')
rgb_test = test_generator_color.flow(x_testRGB, y_testRGB, batch_size=batch_size)
```

Figure 4. Codes to prepare train, validation and test dataset for the RGB and Grayscale channels

Models

Five different models were trained using Adam optimizer with a learning rate of 0.001 and cross entropy loss. Subsequently, they were evaluated using test loss, test accuracy, and micro-F1 score. Listed below are the breakdowns of each model.

1. Custom CNN

One of the earliest models we tested for the distracted driver detection problem was a custom convolutional neural network (CNN). The architecture of the model comprised multiple convolutional blocks, each followed by max-pooling layers to down-sample and retain essential information. The final layers include fully connected neurons for classification into distinct distraction classes. All the layers use ReLU as their activation function except for the last fully connected layer which uses a SoftMax activation function. The model takes in inputs of the size set in the previous steps.

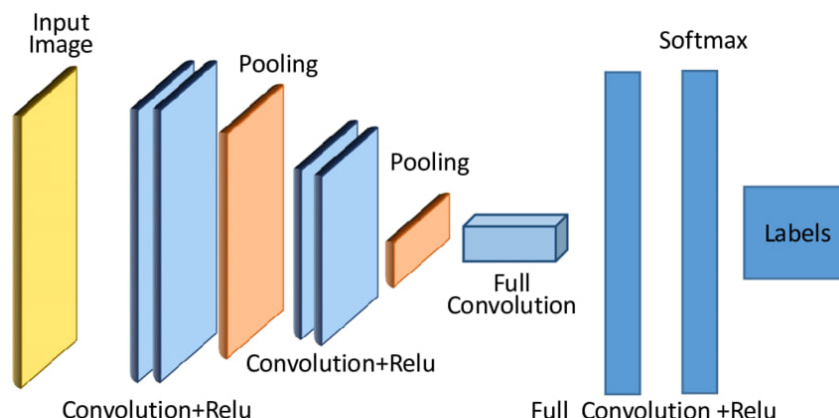


Figure 5. Custom CNN Architecture

Just with this simple model, we were able to achieve a high accuracy of greater than 90 per cent on the validation dataset. However, this accuracy does not transfer over to the test data provided by the competition. The model performed poorly in the test images which was most likely due to overfitting or the model focusing on the wrong features during training.

The development journey involved multiple iterations with different architectures to enhance the model's test accuracy. Techniques such as batch normalization, dropout layers, and fine-tuning of hyperparameters were also employed to try and improve the accuracy. One of the methods we found during research to fix the issue of wrong features being focused is the use of a region of interest (ROI) pooling to focus on the key regions like the head and hands of the driver.

Performance

It had a F1 score on test set of 0.90, a test loss of 0.33 and a kaggle submission loss of 2.51

2. Transfer learning with DenseNet

- DenseNet121 (Densely Connected Convolutional Networks):

DenseNet is a deep learning architecture that introduces the concept of dense connectivity. Unlike traditional convolutional neural networks (CNNs), where each layer is connected only to its subsequent layer, DenseNet connects each layer to all subsequent layers. This dense connectivity pattern facilitates feature reuse across layers, leading to better gradient flow, parameter efficiency, and improved model performance.

In a DenseNet, the output of each layer is concatenated with the inputs of all subsequent layers. This design allows for direct information flow between layers and encourages the model to learn more compact and informative feature representations. DenseNet is known for its parameter efficiency, improved accuracy, and reduced risk of overfitting.

- Transfer Learning

Transfer learning is a machine learning technique where a model trained on one task is adapted to perform a second related task. In the context of deep learning, pre-trained models, which are trained on large datasets for generic tasks, serve as a starting point for training models on specific tasks. Transfer learning is particularly useful when the target dataset is small, as it leverages the knowledge gained from the source task.

There are two main scenarios in transfer learning:

1. Feature Extraction:

In this approach, a pre-trained model is used as a fixed feature extractor. The weights of the early layers are frozen, and only the weights of the final layers (specific to the target task) are trained. This is useful when the low-level features learned by the pre-trained model are relevant to the target task.

2. Fine-Tuning:

In fine-tuning, the entire pre-trained model is further trained on the target task. However, the learning rate is often reduced for the early layers to prevent overfitting and retain the knowledge from the source task. Fine-tuning is suitable when the target task is closely related to the source task, and the target dataset is large enough to avoid overfitting.

- Using Transfer Learning for DenseNet121

Steps for Transfer Learning with DenseNet121:

1. Load Pretrained DenseNet121 Model:

We obtained the pre-trained DenseNet121 model with weights trained on a large dataset, in our case, ImageNet.

2. Modify the Top Layers:

We then Removed the top (fully connected) layers of the pre-trained DenseNet121, and replaced them with layers suitable for the target task, considering the number of classes.

3. Freeze Base Layers:

We also froze the weights of the early layers in DenseNet121 to prevent them from being updated during training. We have done this for all our pre-trained models.

4. Compile and Train

Then we compiled the modified DenseNet121 model with cross entropy loss function, Adam optimizer, and accuracy metric for the task.

By leveraging transfer learning with DenseNet121, we can benefit from the knowledge acquired during the pre-trained model's training on ImageNet while adapting it to a specific task with a smaller dataset. This approach often leads to faster convergence and better generalization performance.

5. Performance

It had a F1 score on test set of 0.76, a test loss of 0.85 and a kaggle submission loss of 2.32

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications import DenseNet121

def DenseNet_Model(img_rows, img_cols, color_type):
    # Load the DenseNet121 model
    base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(img_rows, img_cols, color_type))

    # Freeze the layers in the base model
    for layer in base_model.layers:
        layer.trainable = False

    # Create your model
    model = models.Sequential()
    model.add(base_model)
    model.add(layers.GlobalAveragePooling2D())
    model.add(layers.Dense(512, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dropout(0.25))
    model.add(layers.Dense(NUMBER_CLASSES, activation='softmax'))

    return model

config = tf.compat.v1.ConfigProto()
config.allow_soft_placement = True
tf.compat.v1.keras.backend.set_session(tf.compat.v1.Session(config=config))
```

3. VGG - 16 model

- VGG16 (Visual Geometry Group 16):

VGG16 is a convolutional neural network architecture proposed by the Visual Geometry Group at the University of Oxford. It is characterized by its simplicity and uniformity, with the key design feature being the repeated use of 3x3 convolutional filters. The network consists of 16 weight layers, including 13 convolutional layers and three fully connected layers.

VGG16 is known for its effectiveness in image classification tasks and its straightforward architecture, making it easy to understand and implement. While it may have more parameters than some newer architectures, its simplicity contributes to its widespread use.

- Transfer Learning with VGG16:

Steps for Transfer Learning with VGG16:

1. Load Pretrained VGG16 Model:

We Obtained the pre-trained VGG16 model with weights trained on a large dataset, again, ImageNet.

2. Modify the Top Layers:

We then Removed the top (fully connected) layers of the pretrained VGG16, and replaced them with layers suitable for the target task (10), considering the number of classes.

3. Freeze Base Layers:

We also froze the weights of the early layers in VGG16 to prevent them from being updated during training. We have done this for all our pretrained models.

4. Compile and Train:

Then we compiled the modified VGG16 model with cross entropy loss function, Adam optimizer, and accuracy metric for the task.

5. Performance

It had a F1 score on test set of 0.88, a test loss of 0.53 and a kaggle submission loss of 3.07

```
def VGG16_Model(img_rows, img_cols, color_type=3):
    """
    Architecture and adaptation of the VGG16 for our project
    """
    # Remove fully connected layer and replace
    vgg16_model = VGG16(input_shape=(img_rows, img_cols, color_type), weights="imagenet", include_top=False)
    for layer in vgg16_model.layers:
        layer.trainable = False

    x = vgg16_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(1024, activation='relu')(x)
    predictions = Dense(NUMBER_CLASSES, activation='softmax')(x) # add dense layer with 10 neurons and activation softmax
    model = Model(vgg16_model.input, predictions)
    return model
```

4. ResNet-50

Residual Network with 50 layers (ResNet-50) is a deep convolutional neural network architecture that belongs to the family of ResNet models. It was first introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their paper, “Deep Residual Learning for Image Recognition”. The key innovation of ResNet is the use of residual blocks, otherwise known as skip connections, which enable the direct flow of information from earlier to later layers. This dampens the vanishing gradient problem, and makes it easier to train deeper networks.

```
def ResNet50_Model(img_rows, img_cols, color_type=3):
    resnet_model = ResNet50(input_shape=(img_rows, img_cols, color_type),
                             weights="imagenet", include_top=False)
    for layer in resnet_model.layers:
        layer.trainable = False
    x = GlobalAveragePooling2D()(resnet_model.output)
    x = Dropout(0.5)(x)
    x = Dense(256, activation='relu')(x)

    # Change the output layer to match the number of classes
    predictions = Dense(NUMBER_CLASSES, activation='softmax')(x)

    model = Model(inputs=resnet_model.input, outputs=predictions)
    return model
```

Figure 6. Resnet model code

Atop the base ResNet-50 model, a global average pooling layer was added. Following which, a dropout layer with 0.5 probability to reduce overfitting, and a fully connected layer with 256 units and ReLU activation function were added. Finally, a fully connected layer with 10 units, reflecting the total number of classes, and softmax activation was used to produce the model's predictions.

Performance

It had a F1 score on test set of 0.88, a test loss of 0.39 and a kaggle submission loss of 2.70

5. EfficientNet

About the model:

EfficientNet (Tan & Le, 2020) is a family of convolutional neural network models that is designed to achieve better accuracy and efficiency than previous ConvNets. The models are based on a scaling method that uniformly scales all dimensions of depth, width, and resolution using a compound coefficient.

The scaling method is designed to balance network depth, width, and resolution to achieve better performance. EfficientNet models have achieved state-of-the-art accuracy on various image classification tasks such as ImageNet, CIFAR-100, and Flowers. They are also smaller and faster than previous ConvNets.

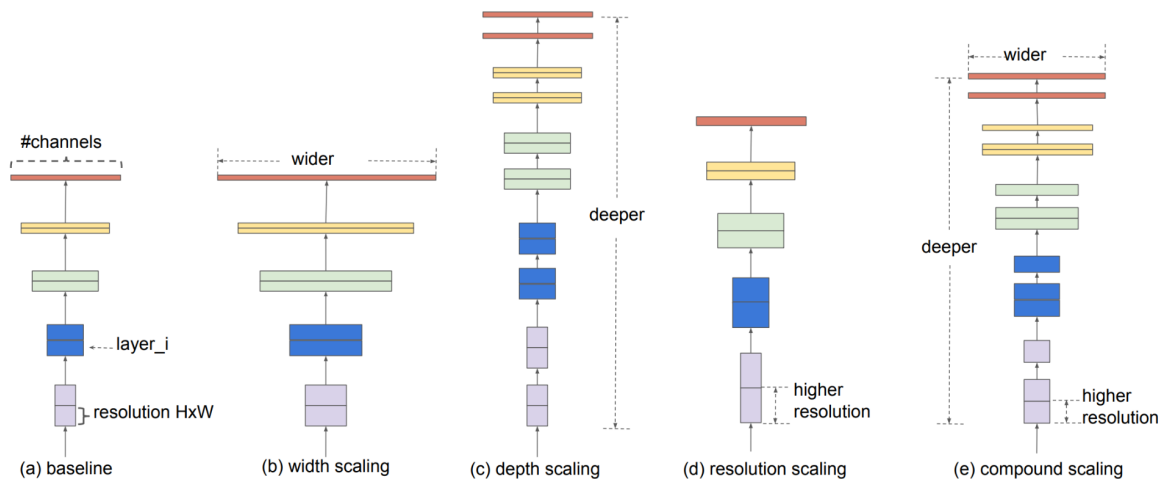


Figure 2. Model Scaling. (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

Figure 7. EfficientNet model visualization

Code:

```
from tensorflow.keras.applications import EfficientNetB3
from tensorflow.keras.layers import Input

def EfficientNet_Model(img_rows, img_cols, color_type):
    inputs = Input(shape=(img_rows, img_cols, color_type))
    base_model = EfficientNetB3(include_top=False, weights='imagenet',
input_tensor=inputs)
    for layer in base_model.layers:
        layer.trainable = False
    x = GlobalAveragePooling2D()(base_model.output)
    x = Dense(512, activation='relu')(x)
```

```

x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(128, activation='relu')(x)
x = Dropout(0.25)(x)
# Change the output layer to match the number of classes
output = Dense(units=NUMBER_CLASSES, activation='softmax')(x)
model = Model(inputs=inputs, outputs=output)
return model

```

1. Inputs and Base Model:

```

def EfficientNet_Model(img_rows, img_cols, color_type):
    inputs = Input(shape=(img_rows, img_cols, color_type))
    base_model = EfficientNetB3(include_top=False, weights='imagenet',
input_tensor=inputs)

```

- `img_rows`, `img_cols`, and `color_type` define the input image dimensions and color channels (e.g., RGB or grayscale).
- `Input(shape=(img_rows, img_cols, color_type))` creates the input layer for the network.
- `EfficientNetB3(include_top=False, weights='imagenet', input_tensor=inputs)` creates the base model using the EfficientNetB3 architecture.
- `include_top=False` indicates to exclude the final classification layer of the pre-trained EfficientNet model.
- `weights='imagenet'` specifies to load pre-trained weights learned on the ImageNet dataset.
- `input_tensor=inputs` specifies the input layer of the network to be the defined `inputs` variable.

2. Feature Extraction and Classification:

```

for layer in base_model.layers:
    layer.trainable = False
x = GlobalAveragePooling2D()(base_model.output)
x = Dense(512, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(128, activation='relu')(x)

```

- `for layer in base_model.layers: layer.trainable = False` freezes the weights of the pre-trained EfficientNet layers. This is done to leverage the features

learned by the pre-trained model while reducing training time and preventing overfitting.

- `GlobalAveragePooling2D()(base_model.output)` performs global average pooling on the output of the base model, reducing the spatial dimensions of the feature maps to a single vector per image.
- `Dense(512, activation='relu')(x)` adds a fully-connected layer with 512 neurons, followed by a Rectified Linear Unit (ReLU) activation function for non-linearity.
- `BatchNormalization()(x)` applies Batch Normalization to improve the stability and convergence of the training process.
- `Dropout(0.5)(x)` introduces dropout with a 50% probability to prevent overfitting.
- `Dense(128, activation='relu')(x)` adds another fully-connected layer with 128 neurons and ReLU activation.
- `Dropout(0.25)(x)` again applies dropout with a lower probability of 25% for further regularization.
- `Dense(units=NUMBER_CLASSES, activation='softmax')(x)` adds the final output layer with the number of neurons equal to the number of classes in the classification task. The softmax activation function normalizes the outputs across classes, resulting in probabilities for each class.

Performance

It had a F1 score on test set of 0.87, a test loss of 0.41 and a kaggle submission loss of 2.03

6. Ensemble Learning

Having independently trained the five models and saved the best performers, the next logical step would be to investigate if their ensemble would yield better results. Firstly, the five models were loaded and used to make predictions on our test data. Next, the results were averaged across all five models.

```
def Ensemble_Model_Predictions_Test(models_list, test_files):
    individual_predictions = []
    for model in models_list:
        pred = model.predict(test_files, batch_size=batch_size, verbose=1)
        individual_predictions.append(pred)
    ensemble_predictions = tf.stack(individual_predictions, axis=0)
    ensemble_predictions = tf.reduce_mean(ensemble_predictions, axis=0)
    return ensemble_predictions
```

Figure 8. Ensemble model code

Following this, the averaged result was passed through a softmax function. Finally, the ensemble was evaluated using test loss, test accuracy, and micro-F1 score.

```
def Eval_Ensemble(y_true, y_pred):
    y_true_sparse = np.argmax(y_true, axis=1)
    cce = tf.keras.losses.SparseCategoricalCrossentropy()
    acc_metric = tf.keras.metrics.CategoricalAccuracy()
    f1_metric = tf.keras.metrics.F1Score(average='micro')

    loss = cce(y_true_sparse, y_pred).numpy()
    acc_metric.update_state(y_true, y_pred)
    acc = acc_metric.result().numpy()
    f1_metric.update_state(y_true, y_pred)
    micro_f1 = f1_metric.result().numpy()

    print("Test Loss:", loss)
    print("Test Accuracy:", acc)
    print("Test micro-f1 Score:", micro_f1)
```

Figure 9. Code to calculate losses

Performance

It had a F1 score on the test set of 0.94, a test loss of 1.65 and a kaggle submission loss of 1.39. Despite the highest test loss, it had the lowest submission loss and thus performed better than the individual models. Based on our results, we declare the ensemble model as the winner.

Results & Evaluation

Kaggle

After preliminary testing, submissions were made to the Kaggle competition which used cross entropy loss as their ranking metric. This was done by loading the best performing models of each variant, followed by obtaining their predictions on the full test suite of 79726 test images, and finally getting their ensemble predictions. The same process was applied to both the grayscale and RGB variants of the models; a total of 12 submissions were made.

```
def Ensemble_Model_Predictions(models_list, test_files, test_targets):
    individual_predictions = []
    for model in models_list:
        pred = model.predict(test_files, batch_size=batch_size, verbose=1)
        individual_predictions.append(pred)
        create_submission(pred, test_targets)
    ensemble_predictions = tf.stack(individual_predictions, axis=0)
    ensemble_predictions = tf.reduce_mean(ensemble_predictions, axis=0)
    return ensemble_predictions
```

Figure 10. Code for Kaggle submission

Submissions are evaluated using the multi-class logarithmic loss. Each image has been labeled with one true class. For each image, you must submit a set of predicted probabilities (one for every image). The formula is then,

$$\text{logloss} = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \log(p_{ij}),$$

Figure 11. Multi-class logarithmic loss formula

where:

- N is the number of images in the test set,
- M is the number of image class labels,
- log is the natural logarithm,
- y_{ij} is 1 if observation i belongs to class j and 0 otherwise,
- p_{ij} is the predicted probability that observation i belongs to class j.

The submitted probabilities for a given image are not required to sum to one because they are rescaled prior to being scored (each row is divided by the row sum). In order to avoid the extremes of the log function, predicted probabilities are replaced with $\max(\min(p, 1 - 10^{-15}), 10^{-15})$.

Listed below are the results of the models tested in Kaggle. As mentioned in the [Data Augmentation](#) section we used photos that were augmented into RGB and Grayscale. Choosing to use grayscale or colored images seemed to have no clear difference in performance across the models.







Submission and Description	Private Score ⓘ	Public Score ⓘ
 ensemble2_rgb.csv Complete (after deadline) · now	1.45898	1.39499
 vgg2_rgb.csv Complete (after deadline) · 20s ago	4.37928	3.82359
 resnet2_rgb.csv Complete (after deadline) · 3m ago	2.39807	2.7977
 enet2_rgb.csv Complete (after deadline) · 3m ago	2.23277	2.026
 dense2_rgb.csv Complete (after deadline) · 3m ago	2.32265	2.11442
 cnn2_rgb.csv Complete (after deadline) · 4m ago	2.5602	2.50591

Figure 12. Kaggle Loss for models using RGB augmented images







Submission and Description	Private Score ⓘ	Public Score ⓘ
 ensemble2_gray.csv Complete (after deadline) · now	1.53174	1.401
 vgg2_gray.csv Complete (after deadline) · 23s ago	3.59286	3.06911
 resnet2_gray.csv Complete (after deadline) · 42s ago	2.57123	2.70704
 enet2_gray.csv Complete (after deadline) · 1m ago	2.194	2.0657
 densenet2_gray.csv Complete (after deadline) · 1m ago	2.15319	2.0369
 cnn2_gray.csv Complete (after deadline) · 2m ago	3.61295	3.14672

Figure 13. Kaggle Loss for models using Grayscale augmented images

Testing

The following is a summary of the best test losses and kaggle scores obtained for all the models. As previously stated, the ensemble model performed the best upon submission despite the highest test loss. More specifically, the ensemble of models trained on colored images performed the best. This is because the ensemble was better at generalizing than the individual models, which could have overfitted.

	Test		Kaggle Test
	F1 Score	Loss	Loss
Custom Grayscale	0.87	0.42	3.15
Custom RGB	0.90	0.33	2.51
DenseNet	0.76 (gray)	0.85(gray)	2.04 (gray)
ResNet50	0.88 (rgb)	0.39 (rgb)	2.70 (gray)
EfficientNet	0.87 (gray)	0.41 (gray)	2.03 (rgb)
VGG-16	0.88 (gray)	0.53 (gray)	3.07 (gray)
Ensemble	0.94 (rgb)	1.65(rgb)	1.39 (rgb)

Figure 14. Test (F1 Score, Loss) and Kaggle (Loss) scores

We thus plot the Confusion Matrix of Ensemble models:

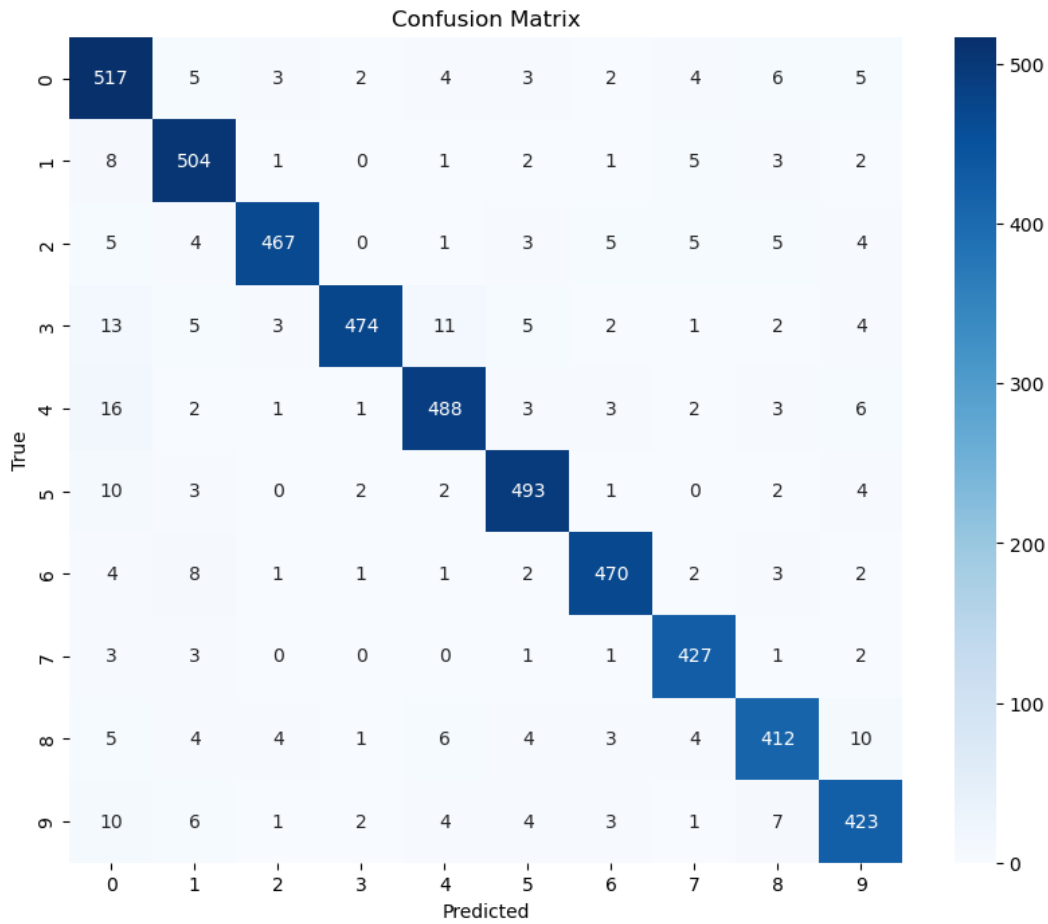


Figure 15. Ensemble Learning confusion matrix

Evidently, most of the images were classified correctly. However, there are several classes that were incorrectly classified as safe driving (class 0) more often than the others. Namely, they are:

- c3: Texting with left hand
- c4: Talking on the phone with left hand
- c5: Operating the radio
- c9: Talking to passenger

For classes 3 and 4, this could be because all of the images were taken from the driver's right. As a result, the left hand may not be entirely visible in many images. While horizontal flipping during image augmentation is one way to address this, it raises another issue - since this process mirrors the images, actions with the right hand become actions with the left, and vice versa. Hence, horizontal flipping may not actually have helped the models learn patterns in the data for this scenario.

For class 5, this could be because the position of the hand operating the radio is close to the steering wheel. Hence, it is more likely to be incorrectly interpreted as being on the steering wheel and indicating safe driving. Meanwhile for class 9,

talking to passengers may involve drivers still having both hands on the steering wheel. This, in turn, is more likely to be incorrectly interpreted as safe driving. To more reliably recognize such patterns, further improvements to the models would have to be made. Gradient visualization may also help us understand how the models are formulating their predictions. This, in turn, would help confirm our suspicions or uncover more insight and improve the training process.

Future Development

As we reflect on our progress, we have identified two key areas as potential improvements for future developments:

1. Region of Interest Pooling (RoI Pooling)

RoI Pooling is a technique commonly used in object detection tasks to extract features from specific regions of an image (K, 2021). In the context of our project, it allows for a more focused extraction of features, specifically honing in on critical areas such as the driver's head and hands within dashboard camera images. This allows the models to capture nuanced details that can potentially further increase the accuracy of our models.

2. Reducing overfitting with advanced techniques

As the issue of overfitting was evident in some of our models, we can further bolster the models' generalization capabilities to reduce overfitting using a comprehensive approach involving advanced techniques:

- Batch Normalization

Batch Normalization is a technique used to improve the stability and speed of training deep neural networks. It operates by normalizing the input of each layer in a mini-batch to have zero mean and unit variance (Doshi, 2019). This normalization, in turn, helps in stabilizing the training process by mitigating issues related to internal covariate shift. By reducing internal covariate shift, it contributes to a more stable training process, making the model less prone to overfitting. It enhances the adaptability of the model to varying input distributions, leading to improved generalization.

- Enhanced Image Augmentations

Beside basic augmentations we have already performed, augmenting the training dataset with a wider variety of images prevents the model from memorizing specific patterns in the training set. This diversity promotes better generalization, reducing the risk of overfitting to the idiosyncrasies of the training data.

- Dropout Layers

Dropout is a regularization technique where, during training, randomly selected neurons are ignored or "dropped out" (Brownlee, 2019). This prevents the model from relying too heavily on specific neurons and encourages it to learn more robust and generalizable features. By introducing an element of randomness during training, it prevents the model from overfitting to noise in the training data, which helps create a more resilient model by discouraging the reliance on specific features, improving generalization to new, unseen data.

- Hyperparameter Tuning
Hyperparameters are parameters that are set before the training process begins. Hyperparameter tuning involves systematically exploring different values for parameters such as learning rates, regularization strengths, and layer configurations to find the combination that optimizes model performance. By fine-tuning hyperparameters, the model's capacity to capture patterns in the training data is optimized without overfitting. This process, however, involves finding a balance that allows the model to generalize well to new data while still capturing the underlying complexity of the task.

Individual Contributions

All members made satisfactory contributions to the project:

- Swastik Majumdar: DenseNet, VGG-16 models, Report
- Win Tun Kyaw: ResNet-50 model, training and testing all the models, Report
- Ishan Monnappa Kodira: EfficientNet, Report, Slides
- Aditi Kumaresan: Model Evaluation, Report
- Nguyen Thai Huy: Data Augmentation, Report, Final Audit
- Prabhakar Dhilahesh: CNN model, Research on Region focused Training, Report

References

- Amazon Web Service. (1978). What is Hyperparameter Tuning?. Amazon.
<https://aws.amazon.com/what-is/hyperparameter-tuning/#:~:text=Hyperparameter%20tuning%20allows%20data%20scientists,the%20model%20as%20a%20hyperparameter.>
- Brownlee, J. (2019, August 6). *A gentle introduction to dropout for Regularizing Deep Neural Networks*. MachineLearningMastery.com.
<https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/>
- Doshi, K. (2021, May 29). *Batch norm explained visually-how it works, and Why Neural Networks Need it*. Medium.
<https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739>
- K, Sambasivarao. (2021, April 30). *Region of interest pooling*. Medium.
<https://towardsdatascience.com/region-of-interest-pooling-f7c637f409af>
- Madhugiri D. (2022, April 22). *Learn Image Augmentation Using 3 Popular Python Libraries*.<https://www.analyticsvidhya.com/blog/2022/04/image-augmentation-using-3-python-libraries/>
- State Farm distracted driver detection*. Kaggle. (n.d.).
<https://www.kaggle.com/competitions/state-farm-distracted-driver-detection>
- Tan, M., & Le, Q. V. (2020, September 11). *EfficientNet: Rethinking model scaling for Convolutional Neural Networks*. arXiv.org. <https://arxiv.org/abs/1905.11946>

Appendix

Code Link on Google Colab: [Link](#)