Automobile Damage Approximator

Sayen Mayuran, Arian Vares, Yash Patel

Overview

Problem: Create AI system that automatically determines how severe the damage on a car might be given an image.
Should be able to:

- Upload images to app as input
- Classify damage severity by mild, moderate, and severe
- Help people in different fields related to the automotive industry
- Program Steps: data processing, training models, using best model for estimation

Potential Models

ResNet-50 → Efficient and modern model, known for solving problem of vanishing gradient through skipping layers

Custom Model → Classic CNN model useful for datasets of simple images and very lightweight and efficient

VGG16 Model → Older but solid model useful for small datasets and fine tuning, requires heavier computation

Data Processing

- Data contains training folder and validation folder containing minor, moderate and severe images of car damage
- Data is extracted into test and training datasets to be used in models
- The data is assigned to integer labels and given class names based on their folders

```
train_ds = tf.keras.utils.image_dataset_from_directory(
    "data3a/training",
    labels="inferred",
    label_mode="int",
    batch_size=None,
    image_size=(256, 256),
    shuffle=True,
)
```

```
test_ds = tf.keras.utils.image_dataset_from_directory(
    "data3a/validation",
    labels="inferred",
    label_mode="int",
    batch_size=None,
    image_size=(256, 256),
    shuffle=False,
)
```

Data Processing

- Training dataset is split into train and validation through “int(len(train_ds)*15)”
- Takes the first 15% of images as validation dataset and remaining as training

```
val_size = int(len(train_ds) * 0.15)

val_ds = train_ds.take(val_size)
train_ds = train_ds.skip(val_size)
```

Data Processing

- Data requires shuffling and batching, done together on code below
- Prefetch increases computational speed through tensorflow's built in ".prefetch(AUTOTUNE)" function

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds = (train_ds
            .shuffle(1000)
            .batch(32)
            .prefetch(AUTOTUNE)
)

val_ds = (val_ds
            .batch(32)
            .prefetch(AUTOTUNE)
)

test_ds = (test_ds
            .batch(32)
            .prefetch(AUTOTUNE)
)
```

Data Processing

- Data is then augmented through a series of random transformations during training (flipping, rotation, etc.) to prevent overfitting

```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
])

train_ds_n_ug = train_ds_n.map(
    lambda x, y: (data_augmentation(x, training=True), y),
    num_parallel_calls=AUTOTUNE
)
```

ResNet-50

- Is a 50 layer deep CNN model (hence the name ResNet**–50**)
- Uses skips to account for vanishing gradient problem
- Model chosen expects inputs of 256x256 images with RGB values
- Initialized with weights trained from ImageNet

ResNet-50

- Model is frozen, prevents initialized weights to be overwritten by unoptimized ones
- Done using “base_model.trainable = False” as shown below

```
input_shape = (256, 256, 3)
num_classes = 3

base_model = tf.keras.applications.ResNet50(
    include_top=False,
    weights="imagenet",
    input_shape=input_shape
)

base_model.trainable = False
```

ResNet-50

- The model takes inputs and creates a feature map with “x = base_model(inputs, training=False)

```
input_shape = (256, 256, 3)
num_classes = 3

base_model = tf.keras.applications.ResNet50(
    include_top=False,
    weights="imagenet",
    input_shape=input_shape
)

base_model.trainable = False


inputs = layers.Input(shape=input_shape)

# Base
x = base_model(inputs, training=False)
```

ResNet-50

- Next lines create a pooling layer to convert outputted feature map into 1D to feed into the FC layer

```
base_model = tf.keras.applications.ResNet50(
    include_top=False,
    weights="imagenet",
    input_shape=input_shape
)

base_model.trainable = False


inputs = layers.Input(shape=input_shape)

# Base
x = base_model(inputs, training=False)

# --- Strong Classification Head ---
x = layers.GlobalAveragePooling2D()(x)
```

ResNet-50

- Last lines of model refer to fully connected layers, called with “layers.Dense(...)”, parameters are # of neurons & activation *f*
- Dense layers paired with normalization and neuron dropouts to account for overfitting

```
x = layers.Dense(512, activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.4)(x)

x = layers.Dense(256, activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.3)(x)

x = layers.Dense(128, activation="relu")(x)
x = layers.BatchNormalization()(x)
outputs = layers.Dense(num_classes, activation="softmax")(x)

model = models.Model(inputs, outputs)
```

ResNet-50

- Model is then compiled and trained for 50 epochs
- Callbacks created for early stopping of training

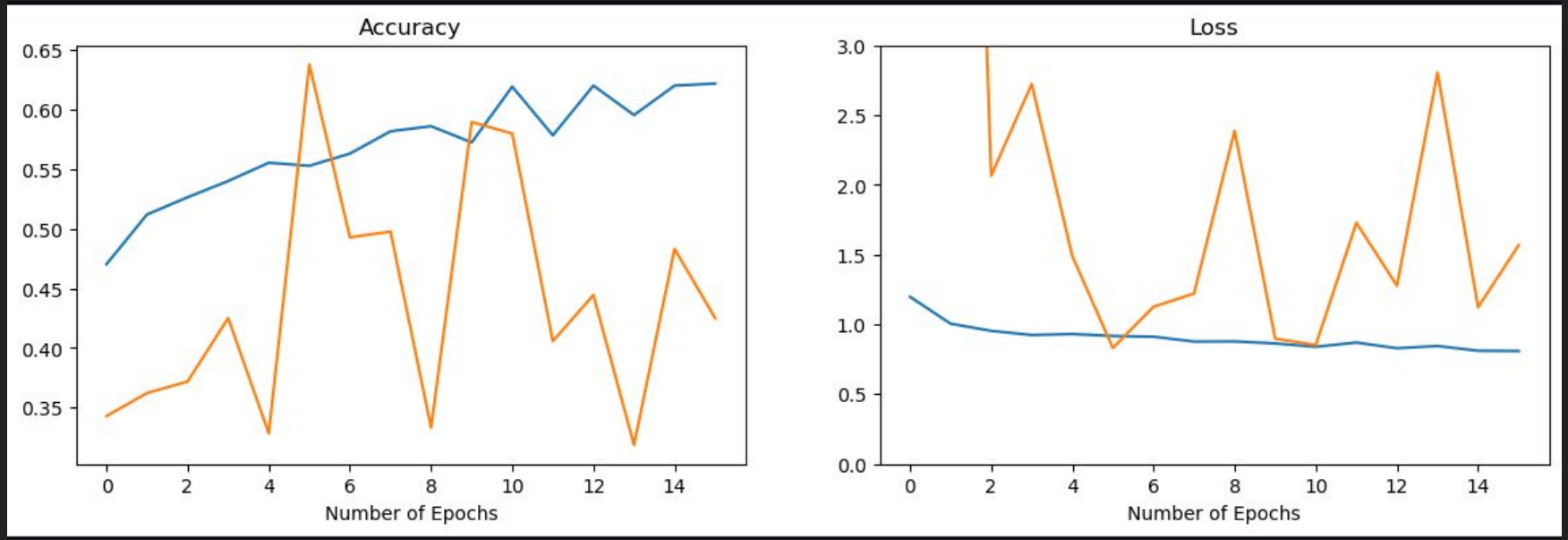
```
model.compile(
    optimizer=tf.keras.optimizers.Adam(0.01),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
```

```
early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss',
                                               patience=10,
                                               restore_best_weights=True)
```

```
history = model.fit(
    train_ds_n,
    validation_data=val_ds_n,
    epochs=50,
    callbacks = [early_stopping]
)
```

ResNet-50

- Results: Underfitting due to small dataset



Custom CNN Model

- Model is classic CNN that utilizes convolution and pooling layers to narrow outputs down to three classifications
- Starts with Convolution layer of 32 filters

Custom Model

```
from tensorflow.keras import layers, models, optimizers

# Define the model
model_custom = models.Sequential()

# Convolutional and pooling layers
model_custom.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)))
```

Custom CNN Model

- Each convolution layer is paired with a pooling layer of “MaxPooling2d…” which downsize each by half, while retaining distinguishing features

```
from tensorflow.keras import layers, models, optimizers

# Define the model
model_custom = models.Sequential()

# Convolutional and pooling layers
model_custom.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)))
model_custom.add(layers.MaxPooling2D((2, 2)))
```

Custom CNN Model

- Each convolution + pooling block uses double the filters of the last beginning with 32 for the first block and 128 for the third
- Blocks followed with “...add(layers.Flatten())” to turn into 1D vector to feed into FC layers

```
# Convolutional and pooling layers
model_custom.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)))
model_custom.add(layers.MaxPooling2D((2, 2)))

model_custom.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_custom.add(layers.MaxPooling2D((2, 2)))

model_custom.add(layers.Conv2D(128, (3, 3), activation='relu'))
model_custom.add(layers.MaxPooling2D((2, 2)))


model_custom.add(layers.Flatten())
```

Custom CNN Model

- FC layers are similar to ResNet-50 where “...layers.Dense” is used with neuron dropping to fight overfitting
- Final FC layer uses “softmax” activation function over “ReLU” for probabilities

```
model_custom.add(layers.Dense(256, activation='relu'))
model_custom.add(layers.Dropout(0.3))

model_custom.add(layers.Dense(128, activation='relu'))
model_custom.add(layers.Dropout(0.3))


model_custom.add(layers.Dense(3, activation='softmax'))
```

Custom CNN Model

- Model is compiled and trained with early stopping callback for 50 epochs

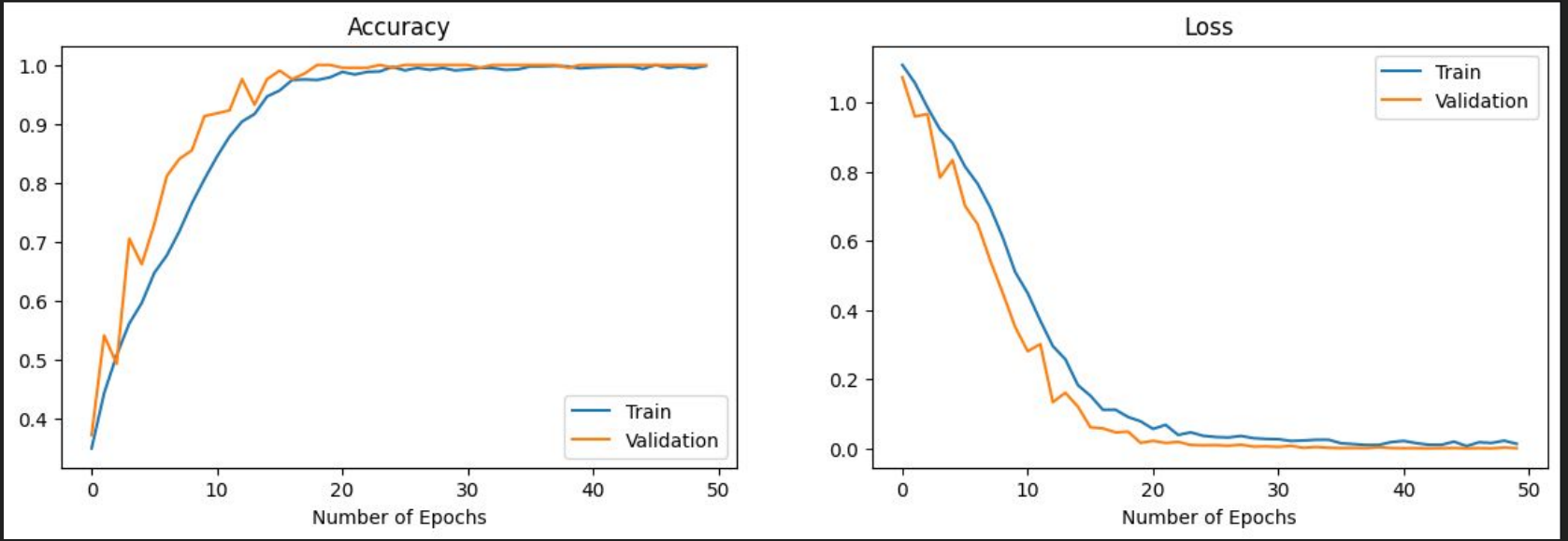
```
# Compile the model
model_custom.compile(optimizer=optimizers.Adam(learning_rate=1e-4),
                     loss='sparse_categorical_crossentropy',
                     metrics=['accuracy'])

epochs = 50

history = model_custom.fit(
    train_ds_n,
    epochs=epochs,
    validation_data=val_ds_n,
    callbacks = [early_stopping]
)
```

Custom CNN Model

- Results: Extremely Accurate



VGG16 Model

- A classic CNN model that only uses 3x3 convolution filters
- Has 16 layers with 13 convolution layers plus 3 fully connected layers
- Initialized with ImageNet weights like ResNet-50 model and frozen like ResNet as well

```
input_shape = (256, 256, 3)
num_classes = 3

base_model = tf.keras.applications.VGG16(
    include_top=False,
    weights="imagenet",
    input_shape=input_shape
)

base_model.trainable = False
```

VGG16 Model

- Inputs are then passed through the model and then used to create a feature map in “x=base_model(inputs,training=False)”
- Pooling layer directly follows the feature map in “GlobalAveragePooling2d()(x)” similar to ResNet50

```
inputs = layers.Input(shape=input_shape)

# Base
x = base_model(inputs, training=False)

# --- Strong Classification Head ---
x = layers.GlobalAveragePooling2D()(x)
```

VGG16 Model

- Fully connected layers follow the convolution and pooling layers, declared with “Dense…” again
- Layers are in a block with normalization and dropout accounting for overfitting,ending with a softmax following the format of the previous models

```
x = layers.Dense(512, activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.4)(x)

x = layers.Dense(256, activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.3)(x)

x = layers.Dense(128, activation="relu")(x)
x = layers.BatchNormalization()(x)
outputs = layers.Dense(num_classes, activation="softmax")(x)
```

VGG16 Model

- Model is similarly compiled and trained for 50 epochs, given the early stopping callback is not called

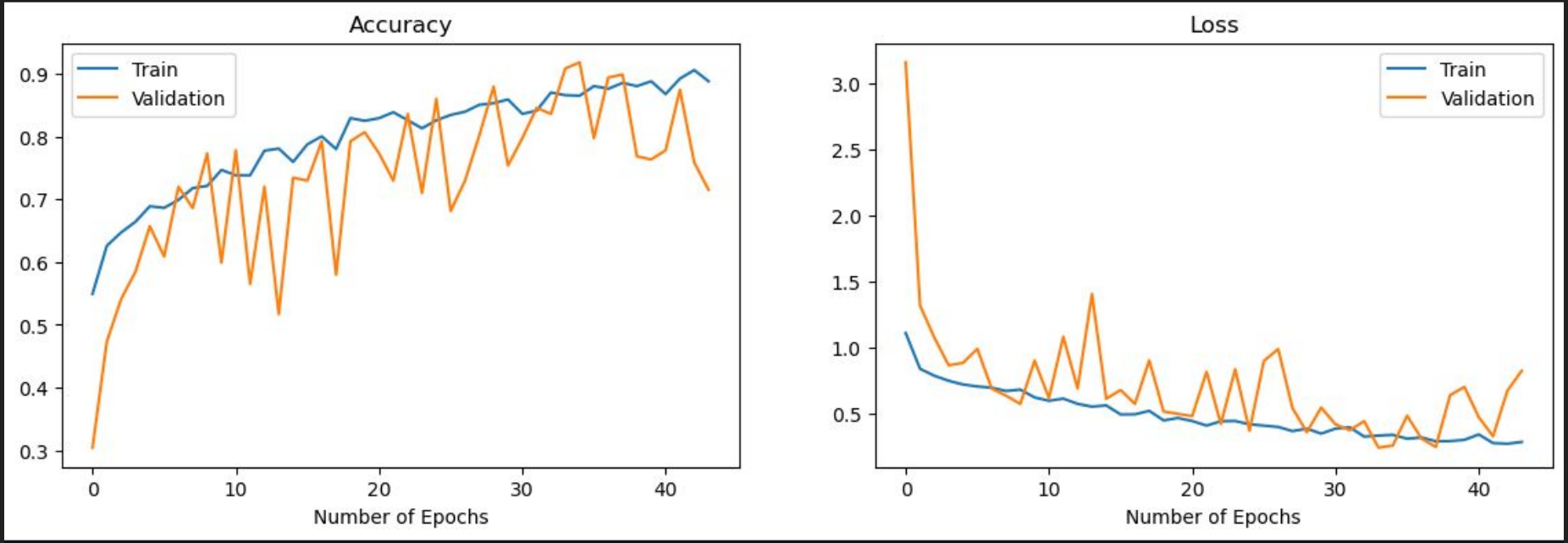
```
early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss',
                                               patience=10,
                                               restore_best_weights=True)
```

```
vgg16.compile(
    optimizer=tf.keras.optimizers.Adam(0.01),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
```

```
history = vgg16.fit(
    train_ds_n,
    validation_data=val_ds_n,
    epochs=50,
    callbacks = [early_stopping]
)
```

VGG16 Model

- Results: Somewhat accurate with slight overfitting



Comparison

Models:	Custom CNN	Frozen VGG16	Frozen ResNet50
Pros	• Stable and smooth without overfitting for both training and validation • Reaches 100% for training and validation	• Smooth increase in training accuracy and increase in validation • Varies greatly between predictions	• Smooth increase for training data
Cons	• Is more intended towards smaller datasets	• Validation data is noisy, not perfect at higher accuracies	• Validation accuracy lowers instead of going up • Not suitable for small datasets as it has lots of depth