NNCV CODED PROJECT

Business Report

DSBA

Submitted By: Maheep Singh
Batch : PGP-DSBA (PGPDSBA.O.Dec24.A)

Table of Contents

List of Figures

List of Tables

No table of figures entries found.

Business Context & Data Dictionary

Context

Workplace safety in hazardous environments like construction sites and industrial plants is crucial to prevent accidents and injuries. One of the most important safety measures is ensuring workers wear safety helmets, which protect against head injuries from falling objects and machinery. Non-compliance with helmet regulations increases the risk of serious injuries or fatalities, making effective monitoring essential, especially in large-scale operations where manual oversight is prone to errors and inefficiency.

To overcome these challenges, SafeGuard Corp plans to develop an automated image analysis system capable of detecting whether workers are wearing safety helmets. This system will improve safety enforcement, ensure compliance and reduce the risk of head injuries. By automating helmet monitoring, SafeGuard aims to enhance efficiency, scalability, and accuracy, ultimately fostering a safer work environment while minimizing human error in safety oversight.

Objective

As a data scientist at SafeGuard Corp, you are tasked with developing an image classification model that classifies images into one of two categories:

- **With Helmet:** Workers wearing safety helmets.
- **Without Helmet:** Workers not wearing safety helmets.

Data Description

The dataset consists of **631 images**, equally divided into two categories:

- **With Helmet:** 311 images showing workers wearing helmets.
- **Without Helmet:** 320 images showing workers not wearing helmets.

Dataset Characteristics:

- **Variations in Conditions:** Images include diverse environments such as construction sites, factories, and industrial settings, with variations in lighting, angles, and worker postures to simulate real-world conditions.
- **Worker Activities:** Workers are depicted in different actions, such as standing, using tools, or moving, ensuring robust model learning for various scenarios.

Rubric Question 1: Exploratory Data Analysis and Data Overview

Data Overview

- **Load Labels**. Below is the information about Label dataset: -

```
RangeIndex: 631 entries, 0 to 630
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Label   631 non-null    int64
dtypes: int64(1)
```

```
(631, 1)
```

Figure 1: Shape of Label & Data Information

- **Load the Images**. Below is the shape of Image Dataset & a random image: -

```
(631, 200, 200, 3)
```

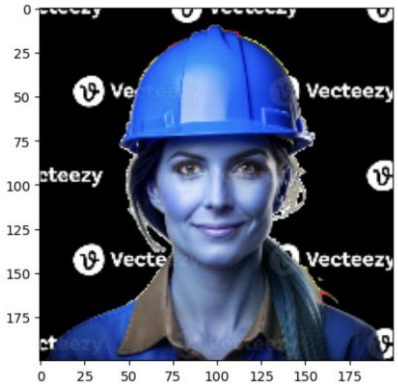


Figure 2: Shape of Images & Random Image from Dataset

- **Converting Images to RGB** format using **cvtColor** function**.** Below is the same random image, post BGR→RGB conversion: -

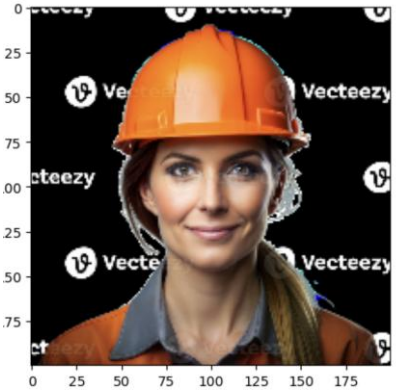


Figure 3: Random Image post BGR→RGB Conversion

Random Image-Label Plot

- Below is a **plot of random images** from each class and with their **corresponding labels**: -

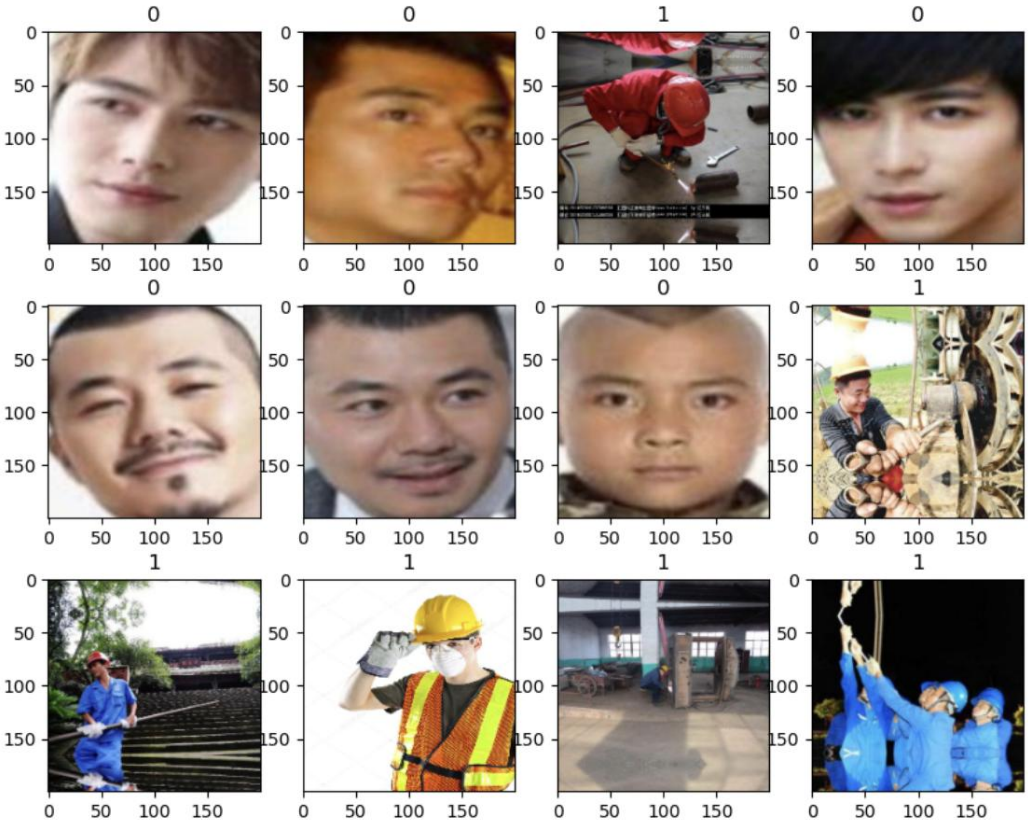
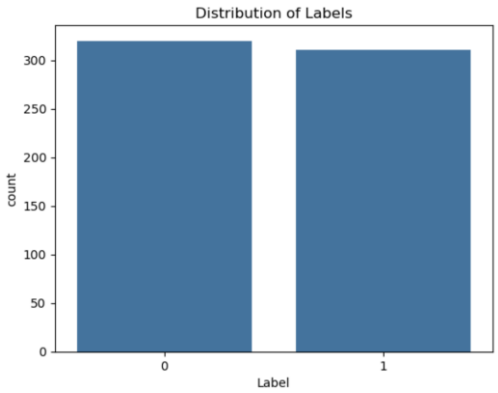


Figure 4: Plot of Random Images & their Labels

Class-Imbalance Check

- **Checking Class Imbalance**: Below is the **CountPlot** showing the **distribution** of both classes in the Label dataset: -




```
Label
0    320
1    311
```

Figure 5: Checking Class Imbalance

Key Meaningful Observations

- **Key Observations: -**
  - There are total of **631 labels** in the Label dataset. **No missing values** found.
  - Correspondingly, **631 Images** in the Image dataset of **size 200 x 200** (image size in pixels) with **3 channels (BGR).** Later the same is converted to RGB format for further analysis.
  - The Label dataset has **2 values: 1→ With Helmet (311 values) | 0→ Without Helmet (320 values).**
  - **No Class Imbalance** observed as both classes have almost similar value counts.

Rubric Question 2: Data Preprocessing

Apply Image Pre-processing Techniques

- **Color-conversion** is often an important image pre-processing step before passing images to Deep Learning models, in order **to simplify their learning tasks** or improve their performance on those tasks.
- We can convert colors in an image using the cvtColor() function of OpenCV.
- Images can be converted to various color schemes, such as Grayscale, Gaussian blur, Laplacian Filter, etc

Grayscale

- Grayscale conversion, helps in **reducing the complexity of the image (one channel instead of three)** and the computational power required to learn from the image, and is hence usually a good idea for Deep Learning models.
- Below is the **GridPlot** of random image, **pre-n-post applying Grayscale**: -

Figure 6: Plot showing Original and Processed Image side by side | Grayscale

- As evident above, the image is converted to black-n-white, reducing the channels from 3 to 1, while keeping the important information like the edges intact.

Gaussian Blur

- **Gaussian Blurring** is a filtering technique for images, where a kernel with values in a Gaussian distribution are used. The values are generated by a Gaussian function.
- It helps in **Noise Removal** and in **smoothening the image, removal of low intensity edges.**
- Below is the **GridPlot** of random image, **pre-n-post applying Gaussian Blur** (kernel size 3x3): -

Figure 7: Plot showing Original and Processed Image side by side | Gaussian Blur

- As evident above, the image edges have been blurred. This helps reduce noise by keeping only the important edges intact, keeping necessary information intact.

Laplacian Filter

- **Laplacian Filter** works by **highlighting those regions where the gradient of pixel intensities changes abruptly.**
- It uses a **single kernel (filter)** to **detect Horizontal & Vertical edges** simultaneously.
- Below is the **GridPlot** of random image, **pre-n-post applying Laplacian Filter** (CV_64F, i.e. output image depth of 64-bit floating point): -

Figure 8: Plot showing Original and Processed Image side by side | Laplacian Filter

- As evident above, the filter is somewhat able to detect the edges in the image (horizontal & vertical). In this case, it gives an impression of the contours of the face in the image.

Splitting the Dataset

- As we have **less images in our dataset** (631 only), we will only use **10% data for testing, 10% data for validation** and the remaining **80% data for training**.
- **Train Dataset** is used to train the model, **Validation Dataset** is used for model evaluation during training, overfitting detection & hyperparameter tuning. **Test Dataset** is used to carry out the final testing of the model as unseen data.

Splitting Dataset – Original Dataset

- We will use **train_test_split()** function from scikit-learn to split the **Original Dataset** into three parts – **Train (80%), Validation (10%) & Test (10%).**
- Below is the **shape** of Train, Validation & Test Datasets: -

```
X_train Shape: (504, 200, 200, 3) | y_train Shape: (504, 1)
X_val Shape: (63, 200, 200, 3) | y_val Shape: (63, 1)
X_test Shape: (64, 200, 200, 3) | y_test Shape: (64, 1)
```

Figure 9: Shape of Original Dataset post Data Splitting

Splitting Dataset – Pre-processed Dataset

- We will use **train_test_split()** function from scikit-learn to split the each of the three **Pre-processed Datasets** into three parts – **Train (80%), Validation (10%) & Test (10%).**
- Below is the **shape** of Train, Validation & Test Datasets: -

```
X_train_gray Shape: (504, 200, 200) | y_train_gray Shape: (504, 1)
X_val_gray Shape: (63, 200, 200) | y_val_gray Shape: (63, 1)
X_test_gray Shape: (64, 200, 200) | y_test_gray Shape: (64, 1)
```

Figure 10: Shape of Pre-processed Dataset post Data Splitting | Grayscale

```
X_train_blur Shape: (504, 200, 200, 3) | y_train_blur Shape: (504, 1)
X_val_blur Shape: (63, 200, 200, 3) | y_val_blur Shape: (63, 1)
X_test_blur Shape: (64, 200, 200, 3) | y_test_blur Shape: (64, 1)
```

Figure 11: Shape of Pre-processed Dataset post Data Splitting | Gaussian Blur

```
X_train_edge Shape: (504, 200, 200) | y_train_edge Shape: (504, 1)
X_val_edge Shape: (63, 200, 200) | y_val_edge Shape: (63, 1)
X_test_edge Shape: (64, 200, 200) | y_test_edge Shape: (64, 1)
```

Figure 12: Shape of Pre-processed Dataset post Data Splitting | Laplacian Filter

Data Normalization

- We carry out data normalization by **dividing the pixels of all images by 255.**
- This is done for the following reasons: -
  - **Pixel Range Standardization**: Raw image pixels are usually in the range [0, 255] (for 8-bit images). Dividing by 255 scales them to [0, 1].
  - **Faster and Stable Training**: Neural networks (especially with activation functions like ReLU or sigmoid) train more efficiently when input values are on a standard, small scale & helps prevent exploding/vanishing gradients and speeds up convergence.
  - **Improved Numerical Stability**: Neural nets work best when the input data is centered or normalized, as large input values can lead to unstable weights and poor performance.
  - **Consistency Across Datasets**: Normalizing ensures that different images (with different lighting or intensity) are on the same scale, which improves generalization.
- **Image Datasets (Original & Pre-processed images) are normalized** by dividing the pixel values of all images by 255, as a pre-requisite to model training.
- Below is a sample of pixel values from Training Dataset (Original Data), before & after data normalization: -

```
X_train
 [[[[ 92  91  97]
   [ 89  88  93]
   [ 54  54  59]
   ...
   [219 217 205]
   [225 224 212]
   [239 239 225]]
```

```
X_train_normalized
 [[[[0.36078432 0.35686275 0.38039216]
   [0.34901962 0.34509805 0.3647059 ]
   [0.21176471 0.21176471 0.23137255]
   ...
   [0.85882354 0.8509804  0.8039216 ]
   [0.88235295 0.8784314  0.83137256]
   [0.9372549  0.9372549  0.88235295]]
```

Figure 13: Pixel Value Sample of Train Dataset, pre-n-post Data Normalization

Label Encoding

- Convert labels to **One Hot Vectors.** This step **may not be required** as the labels are **already binary values (0/1).** However, for **consistency**, we may still go ahead and **convert them to vectors**. LabelBinarizer() Function cane be used for the conversion.
- **Target Label Datasets (Original & Pre-processed Datasets) are encoded.** This shall not impact the overall shape of the datasets.
- Below is the shape of Target Label Datasets (Train, Validation, Test), post Label-Encoding: -

```
Shape of Encoded Target Variable (Original Dataset):
((504, 1), (63, 1), (64, 1))

Shape of Encoded Target Variable (Grayscale Dataset):
((504, 1), (63, 1), (64, 1))

Shape of Encoded Target Variable (Gaussian Blur Dataset):
((504, 1), (63, 1), (64, 1))

Shape of Encoded Target Variable (Laplacian Filter Dataset):
((504, 1), (63, 1), (64, 1))
```

Figure 14: Shape of Encoded Target Variable of Datasets (Train, Validation & Test)

Rubric Question 3: Model building

Model Evaluation Criteria

- Before finalizing the evaluation criteria, lets briefly outline all the evaluation metrics.

Evaluation Metrics Explained

1. **Accuracy:**
   - Definition: Proportion of total correct predictions (both helmet and no-helmet) out of all predictions.
   - In Context: If model predicts 95 out of 100 images correctly, it has 95% accuracy.
   - Limitation: Not useful if the classes are imbalanced as it doesn't tell what kind of mistakes are being made — missing a helmet violation is riskier than a false alarm. In our case this may not be the case as the classes are balanced.
   - High accuracy can mask poor performance on the minority class (in case of class imbalance).
2. **Recall (Sensitivity):**
   - Definition: Proportion of actual "Without Helmet" images correctly identified by the model.
   - In Context: It tells how many unsafe workers the model successfully detects.
   - High Recall: Fewer unsafe workers missed.
   - Low recall: Real safety risks go unnoticed. These are false negatives (worker is not wearing a helmet, but model says they are).
3. **Precision:**
   - Definition: Proportion of predicted "Without Helmet" images that are actually correct.
   - In Context: It tells how many of the flagged violations are truly helmet violations.
   - High precision: Fewer false accusations, which avoids unnecessary alerts or disciplinary actions.
   - Low precision: It may overload safety supervisors with false alarms.
4. **F1-Score:**
   - Definition: Harmonic mean of Precision and Recall — balances the trade-off.
   - In Context: F1-score is useful if we want balanced performance — not missing violations and not raising too many false alarms.
5. **Confusion Matrix:**
   - Definition: A 2x2 matrix showing: -
     - ✓ True Positives (TP): No helmet correctly identified
     - ✓ True Negatives (TN): Helmet correctly identified
     - ✓ False Positives (FP): Helmet worn but flagged as no helmet
     - ✓ False Negatives (FN): No helmet but not detected
   - In Context: The confusion matrix helps to see the actual impact of each kind of error: -
     - ✓ FN (False Negatives) are critical because they mean unsafe behaviour goes undetected.
     - ✓ FP (False Positives) cause minor issues (false alerts), but not safety risks.

Metrics Finalized for Model Evaluation

1. **Primary Metric: Recall**
   - In a **safety-critical application**, **missing a worker not wearing a helmet (false negative) can be dangerous**, leading to safety violations & injuries.
   - Goal of the Image Classification model should be to **Maximize detection of safety violations** in context of the business, rather than someone who is actually compliant (False Positives). Hence, Recall takes precedence over Precision in the business context.
2. **Supplementary Metric: Accuracy**
   - Since, we have a **nearly 50-50 split in the classes**, this **metric won't be misleading.**
   - It can be used as a **General Indicator of Performance since classes are balanced**.
   - Even with balanced classes, accuracy doesn't tell what kind of mistakes the model is making, and in a safety-critical application, that matters. That's why, this cannot be used a primary metric.
3. **Visualize Errors: Confusion Matrix**
   - Confusion Matrix can show the no. of errors (False Positives & False Negatives) in the model to aid visualization of errors in the model.

Model 1 – Simple ANN with Image Flattening

Input Characteristics

- We will use the **Original Dataset** to build this model.
- Below are the Input Image Characteristics from the Training Dataset of Images that would be fed as an input to the ANN Model: -

```
Model 1 Image Input Characteristics:

No. of Classes: 2
Image Size: 120000
Image Input Shape: (200, 200, 3)
```

Figure 15: Model 1 Input Image Characteristics

  - Each Image is of size (200 x 200) with 3 channels (RGB)
  - Total Image Size = 200*200*3 = 1,20,000
  - 2 Label Classes: 0,1

Model Architecture

- **Model Layers: -**
  - **Input Layer:** Accepts image of size (200 x 200 x 3) as a 3D array of pixels.
  - **Flatten Layer**: Converts the 3D image to a 1D vector i.e. (200,200,3) → (120000)
  - **Dense Layer 1 (Hidden Layer)**: Adds non-linearity, helps learn spatial features & learns complex patterns using neurons and 'relu' activation.
    - 256 neurons with 'relu' activation and 'he_uniform' to initialize weights
      *('**he_uniform'** is a weight initialization strategy for deep networks using 'relu' or variants of 'relu' as activation functions. It distributes weights evenly between positive and negative, supporting 'relu' well, speeds up training in deep networks & helps avoid vanishing/exploding gradients by maintaining activation variance across layers).*
  - **Dense Layer 2 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 128 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Dense Layer 3 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 64 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Output Layer:** Outputs a probability score between 0 and 1**.**
    - 1 neuron with 'sigmoid' activation.
- Below is the summary of the model: -

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 120000)	0
dense (Dense)	(None, 256)	30,720,256
dense_1 (Dense)	(None, 128)	32,896
dense_2 (Dense)	(None, 64)	8,256
dense_3 (Dense)	(None, 1)	65

Total params: 30,761,473 (117.35 MB)

Trainable params: 30,761,473 (117.35 MB)

Non-trainable params: 0 (0.00 B)

Figure 16: Model 1 Summary

- Calculate Class-weights, to be fed as a parameter to the model for better performance: -

```
Class Weights: {0: 0.984375, 1: 1.0161290322580645}
```

Figure 17: Class Weights

Compile & Fit Model

- We Compile the Model using the following hyperparameters: -
  - **Loss = binary_crossentropy**
    - Cross-entropy is the most commonly used loss function for classification problems. Cross-entropy measures the difference between the true label and the predicted probability.
    - 'binary_crossentropy' is used as the loss function because the model is doing binary classification and ends with a sigmoid output. It's mathematically tailored for this setup and leads to better performance.
  - **Metrics = Accuracy**
    - Accuracy gives a quick sense of overall model correctness and since the dataset is balanced, the results are not misleading (not biased to majority class)
  - **Optimizer = Adam**
    - Adam optimizer is highly recommended for most deep learning tasks, as it combines the benefits of two powerful optimizers: AdaGrad and RMSProp.
    - Adaptive Learning Rates: Adam adjusts the learning rate for each parameter individually, based on how frequently it's updated and helps learn fine-grained image features like helmet edges or head shape more efficiently.
    - Leverage Momentum: Adam uses momentum to accelerate learning in relevant directions.
    - Together, they make learning: Faster, more stable & less sensitive to noisy gradients.
    - Default hyperparameters (learning_rate=0.001) generally perform well and usually there is no need for extensive manual tuning.
    - Adam handles small datasets (like your 631 images) better than SGD, because it adapts faster and doesn't require huge batch sizes.
  - **Epoch = 30**
    - 30 epochs usually give the model enough passes over the entire dataset to learn complex patterns like helmet shapes, lighting, and posture.
    - Too many epochs can lead to overfitting. 32 usually it considered as a safe ceiling.
  - **Batch-size = 32**
    - 32 is a power-of-2, which aligns well with GPU memory and parallel processing, balancing Training speed & Stability of gradient updates.
    - With only 631 images, a batch size of 32 gives about 20 batches per epoch, which is a good granularity for updates. Smaller batches (like 16) slow training; larger ones (like 64+) might overshoot minima or require more memory.
  - **class_weight = <class_weights_calculated_above>**
    - If one class has significantly more samples than the other, a model might learn to just predict the majority class to get high accuracy. To counter this problem, this parameter tells the model to increase the penalty for misclassifying minority class & reduce the penalty for misclassifying majority class. Hence, it helps the model pay more attention to the underrepresented class.
    - In our case, even though the class weights are nearly balanced, it's a good practice to use this hyperparameter when we are using accuracy as one of key metrics.

- Below is the Model output, post Fit: -

```
Epoch 1/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 7s 233ms/step - accuracy: 0.5460 - loss: 25.2606 - val_accuracy: 0.6190 - val_loss: 4.3637
Epoch 2/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 202ms/step - accuracy: 0.8271 - loss: 1.4546 - val_accuracy: 0.8571 - val_loss: 0.5366
Epoch 3/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 196ms/step - accuracy: 0.8088 - loss: 1.2907 - val_accuracy: 0.7778 - val_loss: 1.8123
Epoch 4/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 195ms/step - accuracy: 0.7862 - loss: 2.1725 - val_accuracy: 0.8254 - val_loss: 1.6255
Epoch 5/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 6s 207ms/step - accuracy: 0.8504 - loss: 1.5604 - val_accuracy: 0.7619 - val_loss: 2.6149
Epoch 6/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 195ms/step - accuracy: 0.8296 - loss: 1.6124 - val_accuracy: 0.8413 - val_loss: 0.9550
Epoch 7/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 203ms/step - accuracy: 0.7901 - loss: 2.5852 - val_accuracy: 0.8413 - val_loss: 1.9838
Epoch 8/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 208ms/step - accuracy: 0.8763 - loss: 1.4447 - val_accuracy: 0.8413 - val_loss: 1.2378
Epoch 9/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 182ms/step - accuracy: 0.9219 - loss: 0.6130 - val_accuracy: 0.9206 - val_loss: 0.4878
Epoch 10/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 175ms/step - accuracy: 0.9571 - loss: 0.2917 - val_accuracy: 0.8889 - val_loss: 0.5220
Epoch 11/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 181ms/step - accuracy: 0.9256 - loss: 0.3247 - val_accuracy: 0.8889 - val_loss: 0.6203
Epoch 12/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 199ms/step - accuracy: 0.9285 - loss: 0.3491 - val_accuracy: 0.8730 - val_loss: 0.9318
Epoch 13/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 192ms/step - accuracy: 0.9144 - loss: 0.4197 - val_accuracy: 0.8889 - val_loss: 0.6122
Epoch 14/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 201ms/step - accuracy: 0.9330 - loss: 0.2928 - val_accuracy: 0.9206 - val_loss: 0.3660
Epoch 15/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 205ms/step - accuracy: 0.9704 - loss: 0.1167 - val_accuracy: 0.8889 - val_loss: 0.4145
Epoch 16/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 199ms/step - accuracy: 0.9814 - loss: 0.1191 - val_accuracy: 0.8730 - val_loss: 0.6899
Epoch 17/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 205ms/step - accuracy: 0.9589 - loss: 0.1791 - val_accuracy: 0.8730 - val_loss: 0.4548
Epoch 18/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 194ms/step - accuracy: 0.9438 - loss: 0.3516 - val_accuracy: 0.9048 - val_loss: 0.4895
Epoch 19/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 198ms/step - accuracy: 0.9821 - loss: 0.1120 - val_accuracy: 0.8889 - val_loss: 1.0337
Epoch 20/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 195ms/step - accuracy: 0.8901 - loss: 0.3735 - val_accuracy: 0.8730 - val_loss: 0.9310
Epoch 21/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 209ms/step - accuracy: 0.9202 - loss: 0.2381 - val_accuracy: 0.8095 - val_loss: 1.6631
Epoch 22/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 201ms/step - accuracy: 0.9265 - loss: 0.3634 - val_accuracy: 0.8730 - val_loss: 0.6922
Epoch 23/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 180ms/step - accuracy: 0.9276 - loss: 0.2601 - val_accuracy: 0.8254 - val_loss: 0.7745
Epoch 24/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 169ms/step - accuracy: 0.9339 - loss: 0.2764 - val_accuracy: 0.8889 - val_loss: 0.4263
Epoch 25/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 183ms/step - accuracy: 0.9507 - loss: 0.3282 - val_accuracy: 0.8571 - val_loss: 0.9868
Epoch 26/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 190ms/step - accuracy: 0.9604 - loss: 0.1515 - val_accuracy: 0.9048 - val_loss: 0.5373
Epoch 27/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 199ms/step - accuracy: 0.9923 - loss: 0.0360 - val_accuracy: 0.8889 - val_loss: 0.6096
Epoch 28/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 200ms/step - accuracy: 0.9877 - loss: 0.0349 - val_accuracy: 0.8889 - val_loss: 0.5640
Epoch 29/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 200ms/step - accuracy: 0.9925 - loss: 0.0281 - val_accuracy: 0.9206 - val_loss: 0.3127
Epoch 30/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 201ms/step - accuracy: 0.9700 - loss: 0.0451 - val_accuracy: 0.8889 - val_loss: 0.6621
```

Figure 18: Model 1 Output

Plot – Model Accuracy vs. Epoch

- Below is plot between Model Accuracy & Epoch for both Training & Validation Dataset: -

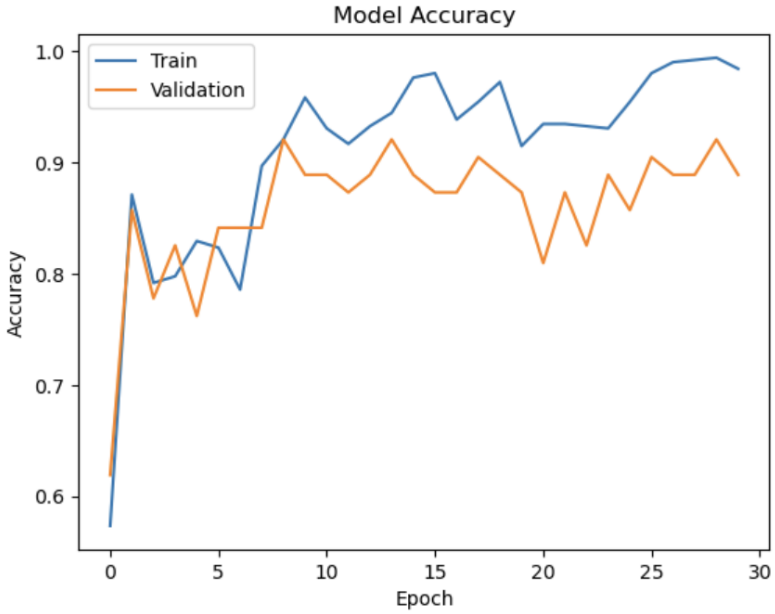


Figure 19: Model 1 | Model Accuracy Plot

- **Key Observations**: -
  - Training accuracy increases steadily, indicating Model learning well on training data
  - Validation accuracy plateaus at around ~10–12 Epochs, meaning Model stops generalizing (learning saturates)
  - Visible gap between Train and Validation accuracy, showing signs of Model overfitting after Epoch 12.
  - Validation Accuracy fluctuations, possibly, signifies that Model is not consistently generalizing.

Model Evaluation

- Below are the Performance Metrics & Confusion Matrices for both Train & Validation Datasets: -

	Accuracy	Precision	Recall	F1 Score
0	0.998016	0.998024	0.998016	0.998016

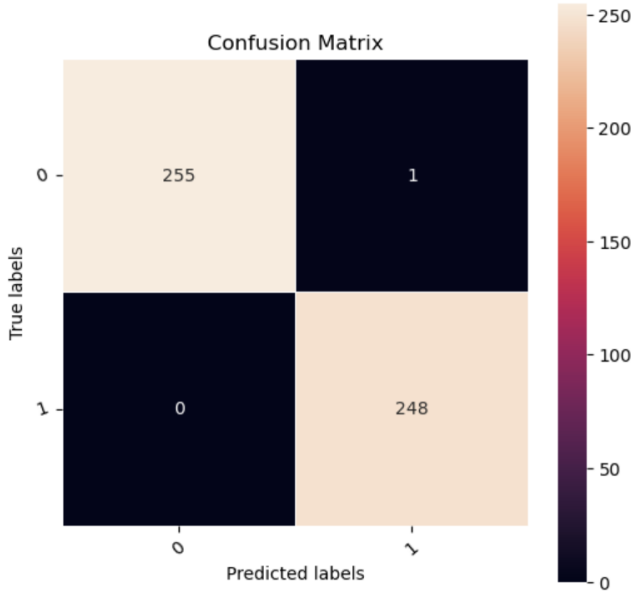


Figure 20: Model 1 Evaluation Metrics | Train

	Accuracy	Precision	Recall	F1 Score
0	0.888889	0.908832	0.888889	0.887295

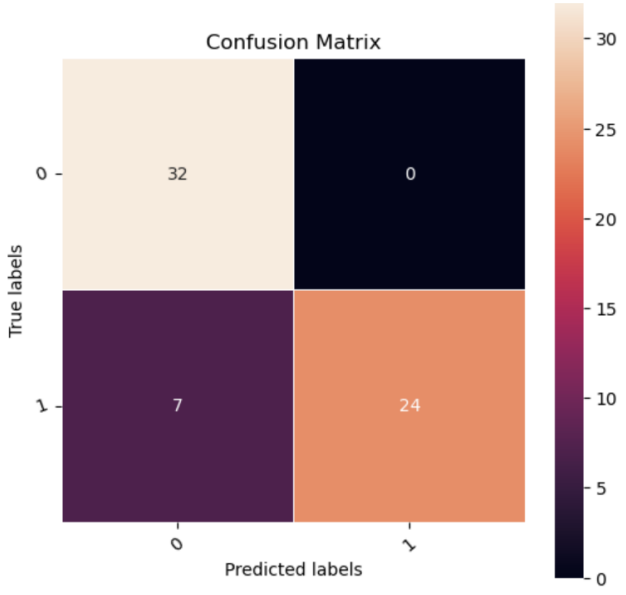


Figure 21: Model 1 Evaluation Metrics | Validation

- **Key Observations: -**
  - **Recall:**
    - Training: 99.80%
    - Validation: 88.89%
  - **Accuracy:**
    - Training: 99.80%
    - Validation: 88.89%
  - **False Negatives ('No Helmet' classified as 'Helmet')**
    - Training: 0 missed violations — ideal for safety.
    - Validation: 7 missed violations — safety concern, must be reduced.
  - **Generalization:** Very high training data metrics, but lagging validation data metrics results reveal less consistent performance on unseen data. This may indicate **overfitting**.
  - **To summarize:** -
    - Model performs **excellently on training data** but **shows signs of overfitting**, especially visible through **false negatives on the validation set**, which are **critical in a safety enforcement context**.
    - There is **scope for improving Recall** from business context standpoint & **reduce overfitting**.

Visualizing Predictions

- Visualizing the predicted and correct label of 3 random images from validation data: -

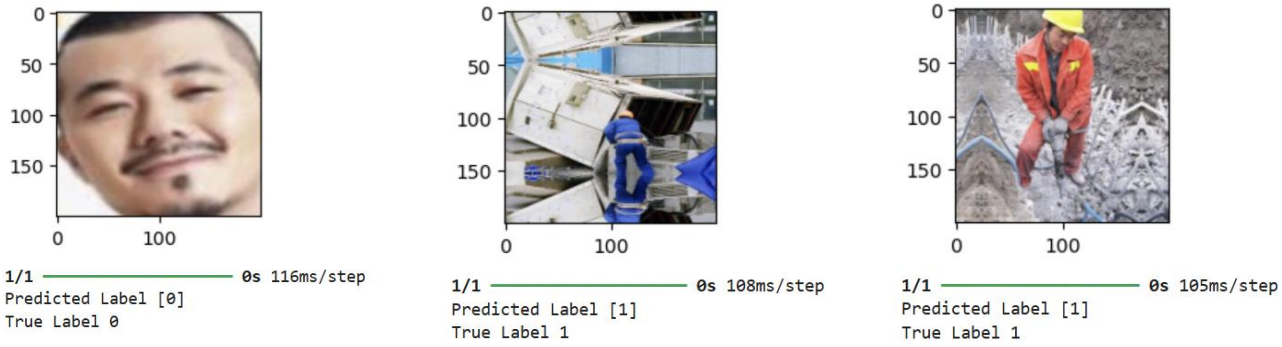


Figure 22: Model 1 – Visualizing Predictions

Model 2 – Simple ANN with Pre-processed Images (Grayscale) and Image Flattening

Input Characteristics

- We will use the **Grayscale Dataset (Pre-processed Images)** to build this model.
- For consistency (and, since 3D vectors are a pre-requisite for CNNs, which would be used in future), lets convert Grayscale Images to a 3D vector by adding channel as the 3rd dimension. We leverage the expand_dims() Function to add a new dimension at the end, converting each image from 2D to 3D by adding a channel dimension., i.e., **(200,200) → (200,200,1)**.
- Below are the Input Image Characteristics from the Training Dataset of Images that would be fed as an input to the ANN Model: -

```
Model 2 Image Input Characteristics:

No. of Classes: 2
Image Size: 40000
Image Input Shape: (200, 200, 1)
(504, 200, 200, 1)
```

Figure 23: Model 2 Input Image Characteristics

  - Each Image is of size (200 x 200) with 1 channel (Grayscale)
  - Total Image Size = 200*200*1 = 40,000
  - 2 Label Classes: 0,1

Model Architecture

- **Model Layers: -**
  - **Input Layer:** Accepts image of size (200 x 200 x 1) as a 3D array of pixels.
  - **Flatten Layer**: Converts the 3D image to a 1D vector i.e. (200,200,1) → (40000)
  - **Dense Layer 1 (Hidden Layer)**: Adds non-linearity, helps learn spatial features & learns complex patterns using neurons and 'relu' activation.
    - 256 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Dense Layer 2 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 128 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Dense Layer 3 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 64 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Output Layer:** Outputs a probability score between 0 and 1**.**
    - 1 neuron with 'sigmoid' activation.
- Below is the summary of the model: -

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 40000)	0
dense (Dense)	(None, 256)	10,240,256
dense_1 (Dense)	(None, 128)	32,896
dense_2 (Dense)	(None, 64)	8,256
dense_3 (Dense)	(None, 1)	65

Total params: 10,281,473 (39.22 MB)

Trainable params: 10,281,473 (39.22 MB)

Non-trainable params: 0 (0.00 B)

Figure 24: Model 2 Summary

Compile & Fit Model

- We Compile the Model using the following hyperparameters: -
  - **Loss = binary_crossentropy**
    - Cross-entropy is the most commonly used loss function for classification problems. Cross-entropy measures the difference between the true label and the predicted probability.
    - 'binary_crossentropy' is used as the loss function because the model is doing binary classification and ends with a sigmoid output. It's mathematically tailored for this setup and leads to better performance.
  - **Metrics = Accuracy**
    - Accuracy gives a quick sense of overall model correctness and since the dataset is balanced, the results are not misleading (not biased to majority class)
  - **Optimizer = Adam**
    - Adam optimizer is highly recommended for most deep learning tasks, as it combines the benefits of two powerful optimizers: AdaGrad and RMSProp.
    - Adaptive Learning Rates: Adam adjusts the learning rate for each parameter individually, based on how frequently it's updated and helps learn fine-grained image features like helmet edges or head shape more efficiently.
    - Leverage Momentum: Adam uses momentum to accelerate learning in relevant directions.
    - Together, they make learning: Faster, more stable & less sensitive to noisy gradients.
    - Default hyperparameters (learning_rate=0.001) generally perform well and usually there is no need for extensive manual tuning.
    - Adam handles small datasets (like your 631 images) better than SGD, because it adapts faster and doesn't require huge batch sizes.
  - **Epoch = 30**
    - 30 epochs usually give the model enough passes over the entire dataset to learn complex patterns like helmet shapes, lighting, and posture.
    - Too many epochs can lead to overfitting. 32 usually it considered as a safe ceiling.
  - **Batch-size = 32**
    - 32 is a power-of-2, which aligns well with GPU memory and parallel processing, balancing Training speed & Stability of gradient updates.
    - With only 631 images, a batch size of 32 gives about 20 batches per epoch, which is a good granularity for updates. Smaller batches (like 16) slow training; larger ones (like 64+) might overshoot minima or require more memory.
  - **class_weight = <class_weights_calculated_above>**
    - If one class has significantly more samples than the other, a model might learn to just predict the majority class to get high accuracy. To counter this problem, this parameter tells the model to increase the penalty for misclassifying minority class & reduce the penalty for misclassifying majority class. Hence, it helps the model pay more attention to the underrepresented class.
    - In our case, even though the class weights are nearly balanced, it's a good practice to use this hyperparameter when we are using accuracy as one of key metrics.

- Below is the Model output, post Fit: -

```
Epoch 1/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 119ms/step - accuracy: 0.5561 - loss: 7.9123 - val_accuracy: 0.8413 - val_loss: 0.4250
Epoch 2/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 88ms/step - accuracy: 0.7618 - loss: 0.9659 - val_accuracy: 0.7619 - val_loss: 0.9483
Epoch 3/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 87ms/step - accuracy: 0.8452 - loss: 0.5001 - val_accuracy: 0.9206 - val_loss: 0.1667
Epoch 4/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 85ms/step - accuracy: 0.8498 - loss: 0.3874 - val_accuracy: 0.8571 - val_loss: 0.3671
Epoch 5/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 88ms/step - accuracy: 0.9121 - loss: 0.2461 - val_accuracy: 0.9206 - val_loss: 0.1439
Epoch 6/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 88ms/step - accuracy: 0.9188 - loss: 0.2269 - val_accuracy: 0.9048 - val_loss: 0.2068
Epoch 7/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 81ms/step - accuracy: 0.9211 - loss: 0.1859 - val_accuracy: 0.9206 - val_loss: 0.1345
Epoch 8/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 84ms/step - accuracy: 0.9419 - loss: 0.1539 - val_accuracy: 0.9206 - val_loss: 0.1346
Epoch 9/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 76ms/step - accuracy: 0.9493 - loss: 0.1617 - val_accuracy: 0.9206 - val_loss: 0.1594
Epoch 10/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 89ms/step - accuracy: 0.8531 - loss: 0.4202 - val_accuracy: 0.7619 - val_loss: 0.8347
Epoch 11/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 92ms/step - accuracy: 0.8250 - loss: 0.6600 - val_accuracy: 0.7619 - val_loss: 0.8585
Epoch 12/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 85ms/step - accuracy: 0.8451 - loss: 0.5793 - val_accuracy: 0.9365 - val_loss: 0.1913
Epoch 13/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 85ms/step - accuracy: 0.9064 - loss: 0.3300 - val_accuracy: 0.9365 - val_loss: 0.2142
Epoch 14/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 89ms/step - accuracy: 0.9063 - loss: 0.3028 - val_accuracy: 0.7302 - val_loss: 1.2274
Epoch 15/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 82ms/step - accuracy: 0.8807 - loss: 0.4885 - val_accuracy: 0.8254 - val_loss: 0.7301
Epoch 16/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 82ms/step - accuracy: 0.9115 - loss: 0.3933 - val_accuracy: 0.7778 - val_loss: 1.0024
Epoch 17/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 85ms/step - accuracy: 0.9079 - loss: 0.3848 - val_accuracy: 0.7937 - val_loss: 0.9888
Epoch 18/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 87ms/step - accuracy: 0.9053 - loss: 0.3726 - val_accuracy: 0.7302 - val_loss: 1.3926
Epoch 19/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 80ms/step - accuracy: 0.9017 - loss: 0.4371 - val_accuracy: 0.6825 - val_loss: 1.9904
Epoch 20/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 88ms/step - accuracy: 0.8611 - loss: 0.6101 - val_accuracy: 0.6825 - val_loss: 3.2638
Epoch 21/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 86ms/step - accuracy: 0.7934 - loss: 1.2638 - val_accuracy: 0.6825 - val_loss: 3.6070
Epoch 22/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 81ms/step - accuracy: 0.7752 - loss: 1.7523 - val_accuracy: 0.9206 - val_loss: 0.2477
Epoch 23/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 83ms/step - accuracy: 0.8455 - loss: 0.9645 - val_accuracy: 0.8413 - val_loss: 1.0647
Epoch 24/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 72ms/step - accuracy: 0.9095 - loss: 0.4648 - val_accuracy: 0.7778 - val_loss: 1.5386
Epoch 25/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 81ms/step - accuracy: 0.8481 - loss: 0.9369 - val_accuracy: 0.9683 - val_loss: 0.2005
Epoch 26/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 82ms/step - accuracy: 0.8414 - loss: 1.0725 - val_accuracy: 0.9206 - val_loss: 0.1821
Epoch 27/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 84ms/step - accuracy: 0.8811 - loss: 0.3364 - val_accuracy: 0.9365 - val_loss: 0.1985
Epoch 28/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 87ms/step - accuracy: 0.9224 - loss: 0.1914 - val_accuracy: 0.9365 - val_loss: 0.2194
Epoch 29/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 86ms/step - accuracy: 0.9191 - loss: 0.2799 - val_accuracy: 0.7460 - val_loss: 1.4806
Epoch 30/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 88ms/step - accuracy: 0.8693 - loss: 0.6747 - val_accuracy: 0.9524 - val_loss: 0.1399
```

Figure 25: Model 2 Output

Plot – Model Accuracy vs. Epoch

- Below is plot between Model Accuracy & Epoch for both Training & Validation Dataset: -

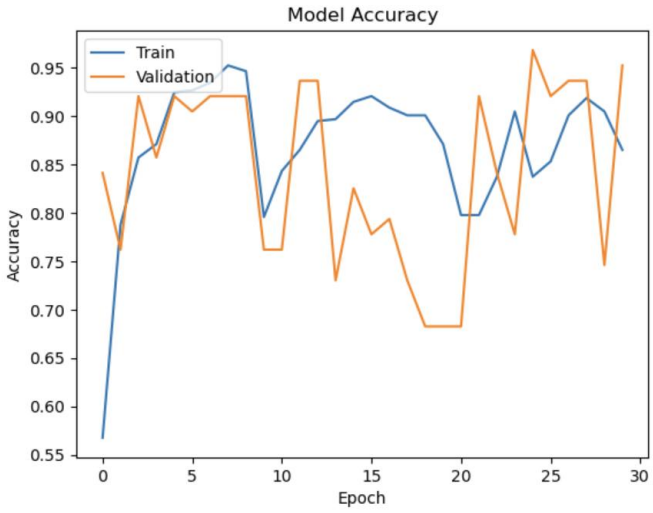


Figure 26: Model 2 | Model Accuracy Plot

- **Key Observations**: -
  - Training accuracy increases steadily from 58% to 95%, then fluctuates between 80-92%, indicating Model learns well initially, but shows instability or noise in later epochs.
  - Validation accuracy starts at 83% & peaks at 96% but shows sharp fluctuations thereafter, suggesting poor Model generalization.
  - Fluctuating gaps between Train and Validation accuracy, showing signs of Model overfitting.
  - It is to be noted that such fluctuations in validation accuracy may be attributed to small validation dataset size (63 samples). Even single misclassification causes swings up to 6%, making validation accuracy curve unstable, even if overall performance is good.

Model Evaluation

- Below are the Performance Metrics & Confusion Matrices for both Train & Validation Datasets: -

	Accuracy	Precision	Recall	F1 Score
0	0.950397	0.952505	0.950397	0.950367

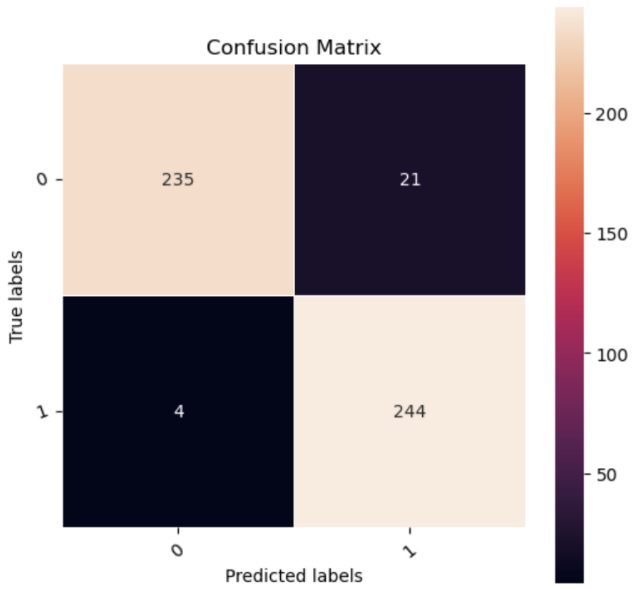


Figure 27: Model 2 Evaluation Metrics | Train

	Accuracy	Precision	Recall	F1 Score
0	0.952381	0.952861	0.952381	0.952381

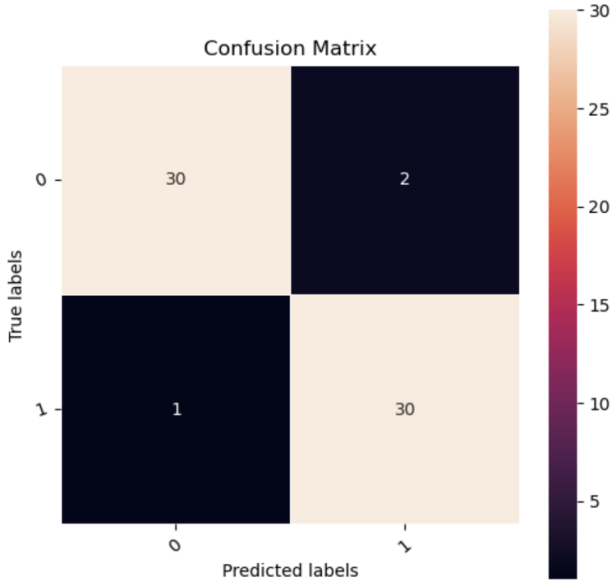


Figure 28: Model 2 Evaluation Metrics | Validation

- **Key Observations: -**
  - **Recall:**
    - Training: 95.04%
    - Validation: 95.24%
  - **Accuracy:**
    - Training: 95.04%
    - Validation: 95.24%
  - **False Negatives ('No Helmet' classified as 'Helmet')**
    - Training: 4 missed violations, increased from previous model
    - Validation: only 1 missed violation, reduced from previous model; suggesting better generalization than previous model
  - **Generalization:** Validation performance is very consistent with training, indicating better generalization.
  - **To summarize:** -
    - Model performs **well equally on training & validation datasets**. Training data performance has reduced, but validation performance improved in comparison to previous model; suggesting **better model-generalization/reduced-overfitting**, while keeping **overall performance fairly good**.

Visualizing Predictions

- Visualizing the predicted and correct label of 3 random images from validation data: -

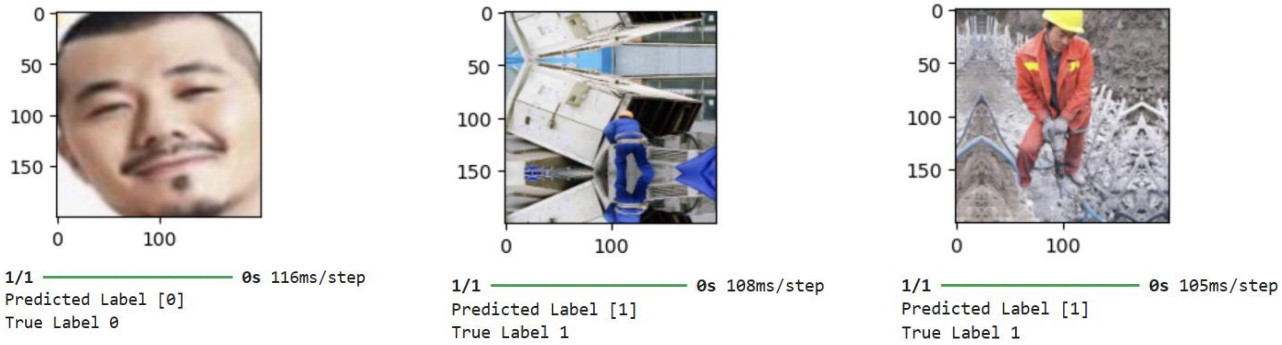


Figure 29: Model 2 – Visualizing Predictions

Model 3 – Basic Convolutional Neural Network with Original Dataset

Input Characteristics

- We will use the **Original Dataset** to build this model.
- Broadly, there are 2 parts to the Model building process: -
  - The **Feature Extraction layers** which are comprised of convolutional and pooling layers.
  - The **Fully Connected Classification** layers for prediction.
- We will also use a **Dropout layer** to reduce overfitting: -
  - Models tend to overfit when we have deep layers in the Neural Network.
  - Prevents overfitting by ensuring the model doesn't rely too heavily on any one feature or neuron.
- Below are the Input Image Characteristics from the Training Dataset of Images that would be fed as an input to the CNN Model: -

```
Model 3 Image Input Characteristics:

No. of Classes: 2
Image Size: 120000
Image Input Shape: (200, 200, 3)
(504, 200, 200, 3)
```

Figure 30: Model 3 Input Image Characteristics

  - Each Image is of size (200 x 200) with 3 channels (RGB)
  - Total Image Size = 200*200*3 = 1,20,000
  - 2 Label Classes: 0,1

Model Architecture

- **Model Layers: -**
  - **Input Layer:** Accepts image of size (200 x 200 x 3) as a 3D array of pixels.
  - **Convolution Layer 1:** Applies 2D filters (kernels) to extract spatial features from images (low-level features like edges) using 32 filters.
    - 32 filters (feature maps) of size (3 x 3) with 'relu' activation & 'same' padding to keep output size same as input.
  - **Max Pooling Layer 1:** Reduces spatial dimensions by half (down-sampling), keeping only the most prominent features.
    - (2x2) pool size that selects the maximum value from each 2×2 window, with 'same' padding to keep the output size equal to input when the window doesn't fit perfectly.
  - **Convolution Layer 2:** Applies 2D filters (kernels) to learn more complex features using 64 filters.
    - 64 filters (feature maps) of size (3 x 3) with 'relu' activation & 'same' padding to keep output size same as input.
  - **Max Pooling Layer 2:** Reduces spatial dimensions by half (down-sampling), keeping only the most prominent features.
    - (2x2) pool size that selects the maximum value from each 2×2 window, with 'same' padding to keep the output size equal to input when the window doesn't fit perfectly.
  - **Convolution Layer 3:** Applies 2D filters (kernels) to extract deeper semantic patterns using 128 filters.
    - 128 filters (feature maps) of size (3 x 3) with 'relu' activation & 'same' padding to keep output size same as input.
  - **Max Pooling Layer 3:** Reduces spatial dimensions by half (down-sampling), keeping only the most prominent features.
    - (2x2) pool size that selects the maximum value from each 2×2 window, with 'same' padding to keep the output size equal to input when the window doesn't fit perfectly.
  - **Dropout Layer:** Randomly deactivates neurons during training to reduce overfitting.
    - 50% of neurons are turned off randomly in this layer during each update.
  - **Flatten Layer**: Converts the 3D image to a 1D vector i.e. (200,200,3) → (120000)
  - **Dense Layer 1 (Hidden Layer)**: Adds non-linearity, helps learn spatial features & learns complex patterns using neurons and 'relu' activation.
    - 256 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Dense Layer 2 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 128 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Dense Layer 3 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 64 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Output Layer:** Outputs a probability score between 0 and 1**.**
    - 1 neuron with 'sigmoid' activation.

- Below is the summary of the model: -

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 200, 200, 32)	896
max_pooling2d (MaxPooling2D)	(None, 100, 100, 32)	0
conv2d_1 (Conv2D)	(None, 100, 100, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 50, 50, 64)	0
conv2d_2 (Conv2D)	(None, 50, 50, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 25, 25, 128)	0
dropout (Dropout)	(None, 25, 25, 128)	0
flatten (Flatten)	(None, 80000)	0
dense (Dense)	(None, 256)	20,480,256
dense_1 (Dense)	(None, 128)	32,896
dense_2 (Dense)	(None, 64)	8,256
dense_3 (Dense)	(None, 1)	65

Total params: 20,614,721 (78.64 MB)

Trainable params: 20,614,721 (78.64 MB)

Non-trainable params: 0 (0.00 B)

Figure 31: Model 3 Summary

Compile & Fit Model

- We Compile the Model using the following hyperparameters: -
  - **Loss = binary_crossentropy**
    - Cross-entropy is the most commonly used loss function for classification problems. Cross-entropy measures the difference between the true label and the predicted probability.
    - 'binary_crossentropy' is used as the loss function because the model is doing binary classification and ends with a sigmoid output. It's mathematically tailored for this setup and leads to better performance.
  - **Metrics = Accuracy**
    - Accuracy gives a quick sense of overall model correctness and since the dataset is balanced, the results are not misleading (not biased to majority class)
  - **Optimizer = Adam**
    - Adam optimizer is highly recommended for most deep learning tasks, as it combines the benefits of two powerful optimizers: AdaGrad and RMSProp.
    - Adaptive Learning Rates: Adam adjusts the learning rate for each parameter individually, based on how frequently it's updated and helps learn fine-grained image features like helmet edges or head shape more efficiently.
    - Leverage Momentum: Adam uses momentum to accelerate learning in relevant directions.
    - Together, they make learning: Faster, more stable & less sensitive to noisy gradients.
    - Default hyperparameters (learning_rate=0.001) generally perform well and usually there is no need for extensive manual tuning.
    - Adam handles small datasets (like your 631 images) better than SGD, because it adapts faster and doesn't require huge batch sizes.
  - **Epoch = 30**
    - 30 epochs usually give the model enough passes over the entire dataset to learn complex patterns like helmet shapes, lighting, and posture.
    - Too many epochs can lead to overfitting. 32 usually it considered as a safe ceiling.

- **Batch-size = 32**
  - 32 is a power-of-2, which aligns well with GPU memory and parallel processing, balancing Training speed & Stability of gradient updates.
  - With only 631 images, a batch size of 32 gives about 20 batches per epoch, which is a good granularity for updates. Smaller batches (like 16) slow training; larger ones (like 64+) might overshoot minima or require more memory.
- **class_weight = <class_weights_calculated_above>**
  - If one class has significantly more samples than the other, a model might learn to just predict the majority class to get high accuracy. To counter this problem, this parameter tells the model to increase the penalty for misclassifying minority class & reduce the penalty for misclassifying majority class. Hence, it helps the model pay more attention to the underrepresented class.
  - In our case, even though the class weights are nearly balanced, it's a good practice to use this hyperparameter when we are using accuracy as one of key metrics.

- Below is the Model output, post Fit: -

```
Epoch 1/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 21s 947ms/step - accuracy: 0.5213 - loss: 1.3630 - val_accuracy: 0.9841 - val_loss: 0.2718
Epoch 2/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 903ms/step - accuracy: 0.9766 - loss: 0.1509 - val_accuracy: 1.0000 - val_loss: 0.0147
Epoch 3/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 911ms/step - accuracy: 0.9945 - loss: 0.0262 - val_accuracy: 0.9524 - val_loss: 0.1052
Epoch 4/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 14s 898ms/step - accuracy: 0.9806 - loss: 0.0579 - val_accuracy: 1.0000 - val_loss: 0.0132
Epoch 5/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 21s 904ms/step - accuracy: 0.9976 - loss: 0.0067 - val_accuracy: 0.9683 - val_loss: 0.0586
Epoch 6/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 14s 898ms/step - accuracy: 0.9987 - loss: 0.0044 - val_accuracy: 0.9683 - val_loss: 0.1599
Epoch 7/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 14s 897ms/step - accuracy: 0.9996 - loss: 0.0048 - val_accuracy: 0.9841 - val_loss: 0.1287
Epoch 8/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 21s 905ms/step - accuracy: 0.9938 - loss: 0.0353 - val_accuracy: 1.0000 - val_loss: 0.0086
Epoch 9/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 905ms/step - accuracy: 0.9931 - loss: 0.0185 - val_accuracy: 0.9365 - val_loss: 0.1490
Epoch 10/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 908ms/step - accuracy: 0.9643 - loss: 0.0941 - val_accuracy: 1.0000 - val_loss: 4.4624e-04
Epoch 11/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 14s 898ms/step - accuracy: 0.9945 - loss: 0.0189 - val_accuracy: 1.0000 - val_loss: 2.3659e-04
Epoch 12/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 21s 902ms/step - accuracy: 0.9989 - loss: 0.0027 - val_accuracy: 1.0000 - val_loss: 5.3474e-04
Epoch 13/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 903ms/step - accuracy: 1.0000 - loss: 4.9725e-04 - val_accuracy: 1.0000 - val_loss: 7.9394e-04
Epoch 14/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 14s 899ms/step - accuracy: 1.0000 - loss: 4.5369e-04 - val_accuracy: 1.0000 - val_loss: 4.1602e-04
Epoch 15/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 904ms/step - accuracy: 1.0000 - loss: 1.8930e-04 - val_accuracy: 1.0000 - val_loss: 3.9528e-05
Epoch 16/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 907ms/step - accuracy: 1.0000 - loss: 1.7738e-05 - val_accuracy: 1.0000 - val_loss: 1.9917e-05
Epoch 17/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 906ms/step - accuracy: 1.0000 - loss: 4.4736e-06 - val_accuracy: 1.0000 - val_loss: 5.3447e-06
Epoch 18/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 14s 900ms/step - accuracy: 1.0000 - loss: 2.0735e-06 - val_accuracy: 1.0000 - val_loss: 3.0224e-06
Epoch 19/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 21s 906ms/step - accuracy: 1.0000 - loss: 2.3369e-06 - val_accuracy: 1.0000 - val_loss: 2.5698e-06
Epoch 20/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 909ms/step - accuracy: 1.0000 - loss: 2.2377e-06 - val_accuracy: 1.0000 - val_loss: 2.4943e-06
Epoch 21/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 903ms/step - accuracy: 1.0000 - loss: 2.1546e-06 - val_accuracy: 1.0000 - val_loss: 2.3157e-06
Epoch 22/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 906ms/step - accuracy: 1.0000 - loss: 4.1102e-06 - val_accuracy: 1.0000 - val_loss: 3.6030e-06
Epoch 23/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 21s 917ms/step - accuracy: 1.0000 - loss: 7.3788e-07 - val_accuracy: 1.0000 - val_loss: 2.7663e-06
Epoch 24/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 14s 899ms/step - accuracy: 1.0000 - loss: 7.3865e-07 - val_accuracy: 1.0000 - val_loss: 1.9488e-06
Epoch 25/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 21s 911ms/step - accuracy: 1.0000 - loss: 1.2712e-06 - val_accuracy: 1.0000 - val_loss: 1.7097e-06
Epoch 26/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 906ms/step - accuracy: 1.0000 - loss: 9.3950e-07 - val_accuracy: 1.0000 - val_loss: 1.4985e-06
Epoch 27/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 907ms/step - accuracy: 1.0000 - loss: 1.0113e-06 - val_accuracy: 1.0000 - val_loss: 1.6192e-06
Epoch 28/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 21s 908ms/step - accuracy: 1.0000 - loss: 2.4476e-07 - val_accuracy: 1.0000 - val_loss: 1.3182e-06
Epoch 29/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 21s 906ms/step - accuracy: 1.0000 - loss: 4.9359e-07 - val_accuracy: 1.0000 - val_loss: 1.0544e-06
Epoch 30/30
16/16 ━━━━━━━━━━━━━━━━━━━━ 15s 910ms/step - accuracy: 1.0000 - loss: 3.0668e-07 - val_accuracy: 1.0000 - val_loss: 8.3850e-07
```

Figure 32: Model 3 Output

Plot – Model Accuracy vs. Epoch

- Below is plot between Model Accuracy & Epoch for both Training & Validation Dataset: -

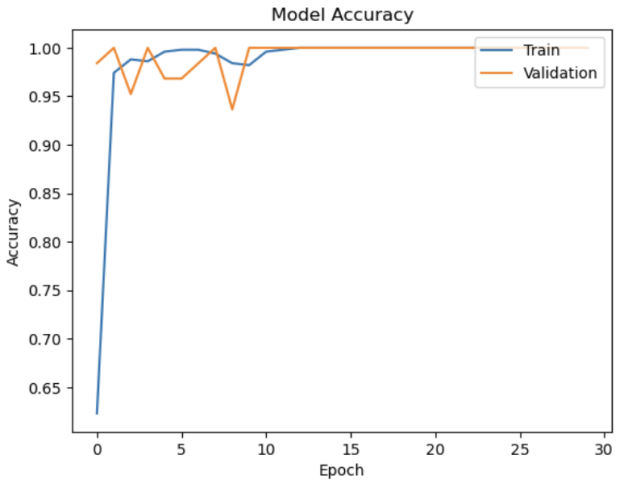


Figure 33: Model 3 | Model Accuracy Plot

- **Key Observations**: -
  - Model learns rapidly in the first few epochs (for both Train & Validation Accuracy), suggesting good feature learning & effective architecture.
  - Training Accuracy reaches 100% by epoch 10, indicating the model is fitting perfectly. There can be a potential of overfitting.
  - Validation accuracy fluctuates at the start but reaches 100% and stays flat, suggesting model generalizes well, but slight instable due to small or noisy validation set,
  - Performance plateaus after 10-12 Epochs for both Training & Validation, indicating model converging early.

Model Evaluation

- Below are the Performance Metrics & Confusion Matrices for both Train & Validation Datasets: -

	Accuracy	Precision	Recall	F1 Score
0	1.0	1.0	1.0	1.0

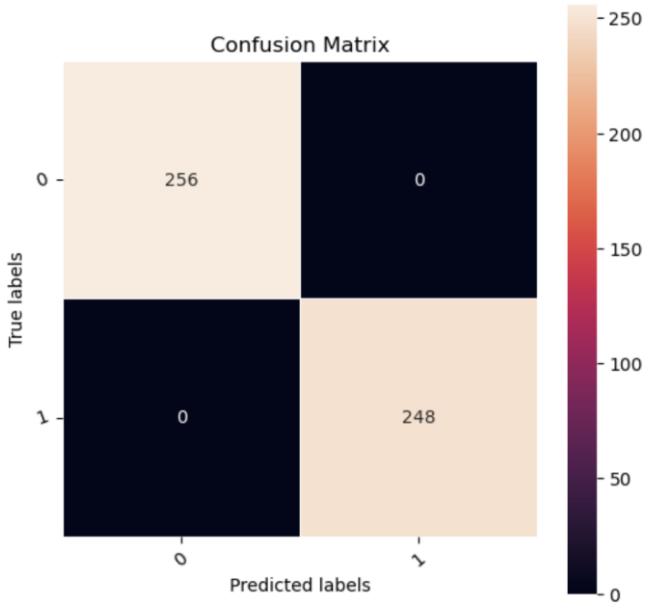


Figure 34: Model 3 Evaluation Metrics | Train

	Accuracy	Precision	Recall	F1 Score
0	1.0	1.0	1.0	1.0

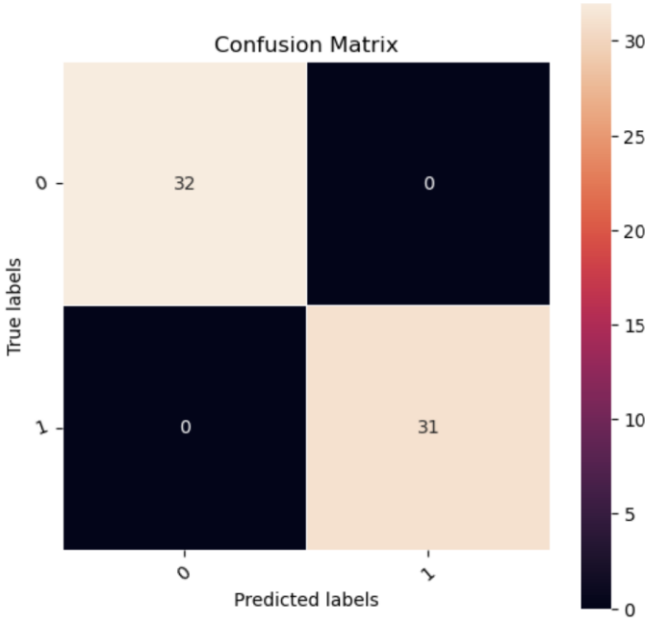


Figure 35: Model 3 Evaluation Metrics | Validation

- **Key Observations: -**
  - **Recall:**
    - Training: 100%
    - Validation: 100%
  - **Accuracy:**
    - Training: 100%
    - Validation: 100%
  - **False Negatives ('No Helmet' classified as 'Helmet')**
    - Training: 0 missed violations, reduction from previous model
    - Validation: 0 missed violation, reduced from previous model
  - **Generalization:** Validation performance is very consistent with training, indicating better generalization.
  - **To summarize:** -
    - Model shows **perfect performance**, with **no errors** on either training or validation sets.
    - It is to be noted that **such results are rare in practice**, so a deeper audit may be required.
    - Due to **small size of the dataset**, **it is easy to overfit**. With only 631 images, especially if they have limited variability (same background, lighting, angles), a model can easily memorize features without learning to generalize. Also, if we are using 10% for validation, that's only 60 images. With such a small set, achieving 100% accuracy may just be luck or overfitting, not true generalization.
    - In the later models, we would augment the data as well to make the model more robust.

Visualizing Predictions

- Visualizing the predicted and correct label of 3 random images from validation data: -

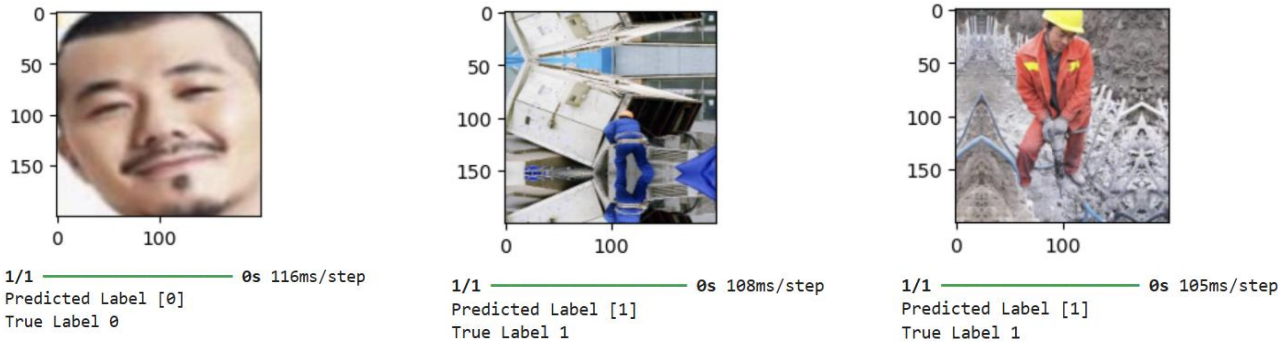


Figure 36: Model 3 – Visualizing Predictions

Model 4 – VGG-16 Model with Feed Forward Neural Network (FFNN) using Original Dataset

Input Characteristics

- We will use the **Original Dataset** to build this model.
- **Transfer Learning** to be leveraged by loading a pre-built architecture, **VGG16**, which was trained on the ImageNet dataset and is the runner-up in the ImageNet competition in 2014.
- We will **directly use the convolutional and pooling layers and freeze their weights** i.e. no training will be done on them. We will remove the already-present fully-connected layers **and add our own fully-connected layers** for this binary classification task.
- For classification, we will add a Flatten layer and a Feed Forward Neural Network.
- We will also use a **Dropout layer** to reduce overfitting: -
  - Models tend to overfit when we have deep layers in the Neural Network.
  - Prevents overfitting by ensuring the model doesn't rely too heavily on any one feature or neuron.
- **Resize input image to (224,224,3)** – VGG16 expects input image of size (224,224,3). So, to use VGG16 model effectively, it requires input image to be resized from **(220,220,3) → (224,224,3).** This shall be done via **Resize() Function with Linear Interpolation.**
- Below are the Input Image Characteristics from the Training Dataset of Images that would be fed as an input to the CNN Model: -

```
Model 4 Image Input Characteristics:

No. of Classes: 2
Image Size: 150528
Image Input Shape: (224, 224, 3)
(504, 224, 224, 3)
```

Figure 37: Model 4 Input Image Characteristics

  - Each Image is of size (224 x 224) with 3 channels (RGB)
  - Total Image Size = 224*224*3 = 1,50,528
  - 2 Label Classes: 0,1

Model Architecture

- **Loading VGG16 Model: -**

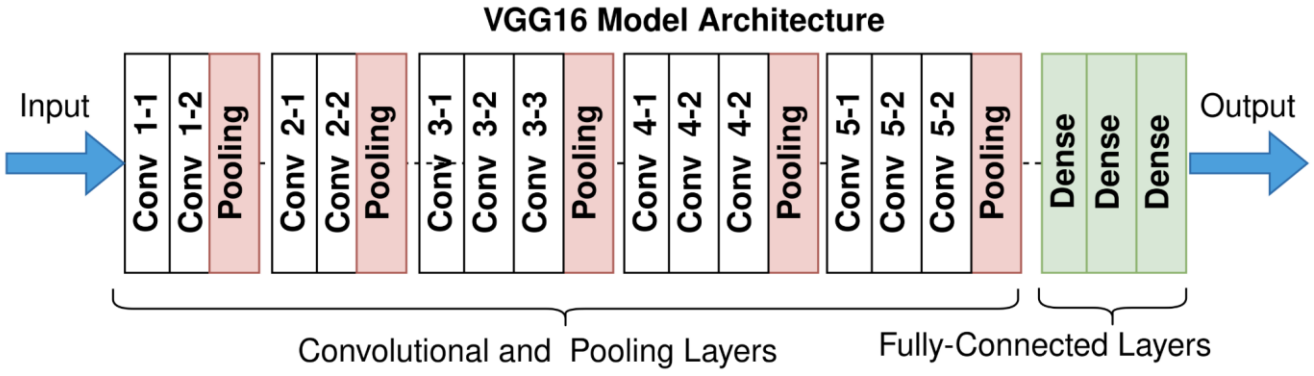


Figure 38: VGG16 Model Architecture

- VGG16 Model, pre-trained model with its weights, is loaded. Layer-types remain the same as described earlier, except, VGG16 has a peculiar architecture as shown above.
- To remove the Fully-connected layers of the imported pre-trained model, we can also specify an additional keyword argument – 'include_top = False'. Then, the model will be imported without the fully-connected layers.
- We won't have to do all the steps of getting the last convolutional layer and creating a separate model.
- Once, the **VGG16 Model (without Fully-connected Layers) is loaded with its weights, we freeze all the layers for training.**

- Below is the **Model Summary** of the **VGG16 Model (without Fully-connected Layers): -**

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0

Total params: 14,714,688 (56.13 MB)

Trainable params: 14,714,688 (56.13 MB)

Non-trainable params: 0 (0.00 B)

Figure 39: VGG16 Model Summary (without Fully-connected Layers)

```
input_layer_1 False
block1_conv1 False
block1_conv2 False
block1_pool False
block2_conv1 False
block2_conv2 False
block2_pool False
block3_conv1 False
block3_conv2 False
block3_conv3 False
block3_pool False
block4_conv1 False
block4_conv2 False
block4_conv3 False
block4_pool False
block5_conv1 False
block5_conv2 False
block5_conv3 False
block5_pool False
```

Figure 40: VGG16 Model (without Fully-connected Layers) – Layer Freeze for Training

- **Final Model Layers: -**
  - **VGG16 Model Layer:** This is the pre-trained convolutional part of VGG16, which extracts high-level image features like edges, textures, and object parts. It doesn't include the dense layers from VGG16's classifier.
    - VGG16 Convolution Layers (without fully-connected layers) with Input Layer which is already fed into VGG16 Model.
  - **Flatten Layer**: Converts the 3D image to a 1D vector i.e. (7,7,512) → (25088)
  - **Dense Layer 1 (Hidden Layer)**: Adds non-linearity, helps learn spatial features & learns complex patterns using neurons and 'relu' activation.
    - 256 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Dense Layer 2 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 128 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Dropout Layer:** Randomly deactivates neurons during training.
    - 50% of neurons are turned off randomly in this layer during each update. **In previous Models we suspected overfitting, so we introduced this layer between the Dense Layers to Regularize the Model.**
  - **Dense Layer 3 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 64 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Output Layer:** Outputs a probability score between 0 and 1**.**
    - 1 neuron with 'sigmoid' activation.
- Below is the summary of the model: -

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 256)	6,422,784
dense_1 (Dense)	(None, 128)	32,896
dropout (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 64)	8,256
dense_3 (Dense)	(None, 1)	65

Total params: 21,178,689 (80.79 MB)

Trainable params: 6,464,001 (24.66 MB)

Non-trainable params: 14,714,688 (56.13 MB)

Figure 41: Model 4 Summary

Compile & Fit Model

- We Compile the Model using the following hyperparameters: -
  - **Loss = binary_crossentropy**
    - Cross-entropy is the most commonly used loss function for classification problems. Cross-entropy measures the difference between the true label and the predicted probability.
    - 'binary_crossentropy' is used as the loss function because the model is doing binary classification and ends with a sigmoid output. It's mathematically tailored for this setup and leads to better performance.
  - **Metrics = Accuracy**
    - Accuracy gives a quick sense of overall model correctness and since the dataset is balanced, the results are not misleading (not biased to majority class)
  - **Optimizer = Adam**
    - Adam optimizer is highly recommended for most deep learning tasks, as it combines the benefits of two powerful optimizers: AdaGrad and RMSProp.
    - Adaptive Learning Rates: Adam adjusts the learning rate for each parameter individually, based on how frequently it's updated and helps learn fine-grained image features like helmet edges or head shape more efficiently.

- Leverage Momentum: Adam uses momentum to accelerate learning in relevant directions.
- Together, they make learning: Faster, more stable & less sensitive to noisy gradients.
- Default hyperparameters (learning_rate=0.001) generally perform well and usually there is no need for extensive manual tuning.
- Adam handles small datasets (like your 631 images) better than SGD, because it adapts faster and doesn't require huge batch sizes.

- **Epoch = 30**
  - 30 epochs usually give the model enough passes over the entire dataset to learn complex patterns like helmet shapes, lighting, and posture.
  - Too many epochs can lead to overfitting. 32 usually it considered as a safe ceiling.
- **Batch-size = 32**
  - 32 is a power-of-2, which aligns well with GPU memory and parallel processing, balancing Training speed & Stability of gradient updates.
  - With only 631 images, a batch size of 32 gives about 20 batches per epoch, which is a good granularity for updates. Smaller batches (like 16) slow training; larger ones (like 64+) might overshoot minima or require more memory.
  - Only applies when training on raw NumPy arrays, not generators. In our case, this will be ignored (refer below section)
- **class_weight = <class_weights_calculated_above>**
  - If one class has significantly more samples than the other, a model might learn to just predict the majority class to get high accuracy. To counter this problem, this parameter tells the model to increase the penalty for misclassifying minority class & reduce the penalty for misclassifying majority class. Hence, it helps the model pay more attention to the underrepresented class.
  - In our case, even though the class weights are nearly balanced, it's a good practice to use this hyperparameter when we are using accuracy as one of key metrics.
- **Standardize Input Flow into the Model**
  - **Converts your NumPy arrays into a batch-generating iterator that feeds data into the model efficiently.**
  - Handles **Shuffling, Batching, and Seeding**: -
    I. **batch_size=128**
    II. **shuffle=True**
    III. **seed=42**
  - ImageDataGenerator.flow() Function is used for the same.

- Below is the Model output, post Fit: -

```
Epoch 1/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 124s 31s/step - accuracy: 0.5722 - loss: 0.9451 - val_accuracy: 0.9524 - val_loss: 0.1955
Epoch 2/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 30s/step - accuracy: 0.9088 - loss: 0.2048 - val_accuracy: 1.0000 - val_loss: 0.0020
Epoch 3/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 119s 31s/step - accuracy: 0.9876 - loss: 0.0438 - val_accuracy: 1.0000 - val_loss: 6.8835e-04
Epoch 4/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 141s 30s/step - accuracy: 0.9966 - loss: 0.0082 - val_accuracy: 1.0000 - val_loss: 1.6088e-04
Epoch 5/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 143s 31s/step - accuracy: 0.9934 - loss: 0.0125 - val_accuracy: 1.0000 - val_loss: 5.7547e-06
Epoch 6/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 30s/step - accuracy: 0.9957 - loss: 0.0074 - val_accuracy: 1.0000 - val_loss: 5.3116e-07
Epoch 7/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 142s 30s/step - accuracy: 0.9963 - loss: 0.0088 - val_accuracy: 1.0000 - val_loss: 2.1384e-07
Epoch 8/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 30s/step - accuracy: 1.0000 - loss: 2.6637e-04 - val_accuracy: 1.0000 - val_loss: 1.2229e-07
Epoch 9/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 31s/step - accuracy: 1.0000 - loss: 0.0011 - val_accuracy: 1.0000 - val_loss: 1.3610e-07
Epoch 10/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 30s/step - accuracy: 1.0000 - loss: 7.2688e-04 - val_accuracy: 1.0000 - val_loss: 2.0352e-07
Epoch 11/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 121s 32s/step - accuracy: 1.0000 - loss: 3.1025e-05 - val_accuracy: 1.0000 - val_loss: 2.7621e-07
Epoch 12/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 123s 30s/step - accuracy: 1.0000 - loss: 8.3703e-05 - val_accuracy: 1.0000 - val_loss: 3.1568e-07
Epoch 13/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 31s/step - accuracy: 1.0000 - loss: 4.2293e-04 - val_accuracy: 1.0000 - val_loss: 3.8326e-07
Epoch 14/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 31s/step - accuracy: 1.0000 - loss: 2.0316e-04 - val_accuracy: 1.0000 - val_loss: 3.8525e-07
Epoch 15/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 30s/step - accuracy: 1.0000 - loss: 1.1816e-04 - val_accuracy: 1.0000 - val_loss: 3.7086e-07
Epoch 16/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 31s/step - accuracy: 1.0000 - loss: 1.4370e-05 - val_accuracy: 1.0000 - val_loss: 3.5649e-07
Epoch 17/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 121s 32s/step - accuracy: 1.0000 - loss: 1.0716e-05 - val_accuracy: 1.0000 - val_loss: 3.4243e-07
Epoch 18/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 120s 32s/step - accuracy: 1.0000 - loss: 2.6458e-04 - val_accuracy: 1.0000 - val_loss: 2.5813e-07
Epoch 19/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 142s 32s/step - accuracy: 1.0000 - loss: 6.0676e-05 - val_accuracy: 1.0000 - val_loss: 1.7263e-07
Epoch 20/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 120s 32s/step - accuracy: 1.0000 - loss: 6.4690e-05 - val_accuracy: 1.0000 - val_loss: 1.2810e-07
Epoch 21/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 120s 31s/step - accuracy: 1.0000 - loss: 0.0011 - val_accuracy: 1.0000 - val_loss: 5.4288e-08
Epoch 22/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 120s 31s/step - accuracy: 1.0000 - loss: 1.0960e-05 - val_accuracy: 1.0000 - val_loss: 2.3085e-08
Epoch 23/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 120s 31s/step - accuracy: 1.0000 - loss: 8.3711e-04 - val_accuracy: 1.0000 - val_loss: 1.4789e-08
Epoch 24/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 120s 31s/step - accuracy: 1.0000 - loss: 1.2192e-06 - val_accuracy: 1.0000 - val_loss: 1.0897e-08
Epoch 25/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 121s 31s/step - accuracy: 1.0000 - loss: 8.4549e-06 - val_accuracy: 1.0000 - val_loss: 8.8808e-09
Epoch 26/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 129s 34s/step - accuracy: 1.0000 - loss: 5.2082e-06 - val_accuracy: 1.0000 - val_loss: 7.7268e-09
Epoch 27/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 136s 32s/step - accuracy: 1.0000 - loss: 6.1823e-05 - val_accuracy: 1.0000 - val_loss: 7.6914e-09
Epoch 28/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 117s 30s/step - accuracy: 1.0000 - loss: 2.5512e-05 - val_accuracy: 1.0000 - val_loss: 9.2806e-09
Epoch 29/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 30s/step - accuracy: 1.0000 - loss: 6.0209e-05 - val_accuracy: 1.0000 - val_loss: 1.0579e-08
Epoch 30/30
4/4 ━━━━━━━━━━━━━━━━━━━━ 118s 31s/step - accuracy: 1.0000 - loss: 3.7667e-05 - val_accuracy: 1.0000 - val_loss: 1.2132e-08
```

Figure 42: Model 4 Output

Plot – Model Accuracy vs. Epoch

- Below is plot between Model Accuracy & Epoch for both Training & Validation Dataset: -

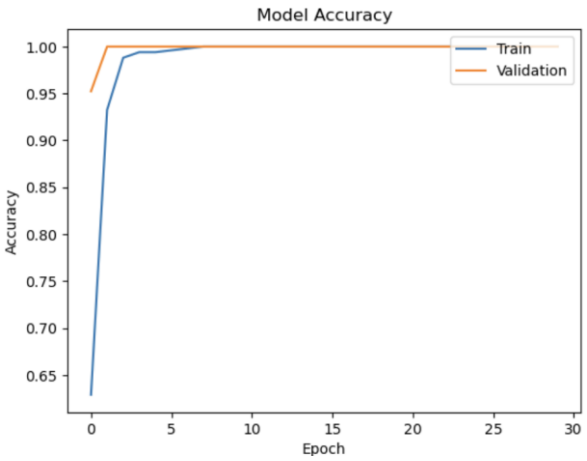


Figure 43: Model 4 | Model Accuracy Plot

- **Key Observations**: -
  - Model learns rapidly in the first few epochs (for both Train & Validation Accuracy), suggesting good feature learning.
  - Training Accuracy rises quickly from 62% to 100% by epoch 5 & stays flat, indicating Model fits training data extremely well, possibly memorizing it.
  - Validation accuracy also reaches 100% within 5 epochs, indicating strong generalization.
  - No gaps observed between training and validation curves, suggesting no overfitting.
  - Performance plateaus after 6-8 Epochs for both Training & Validation, indicating models converged early.

Model Evaluation

- Below are the Performance Metrics & Confusion Matrices for both Train & Validation Datasets: -

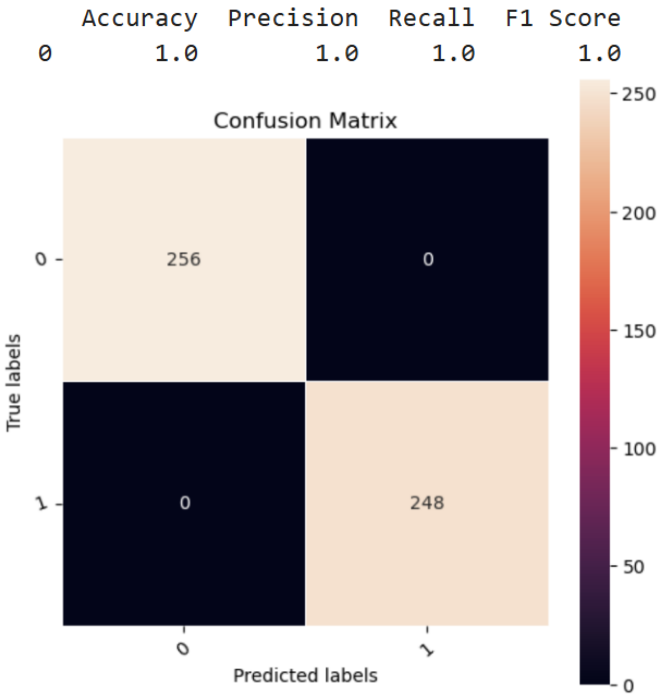


Figure 44: Model 4 Evaluation Metrics | Train

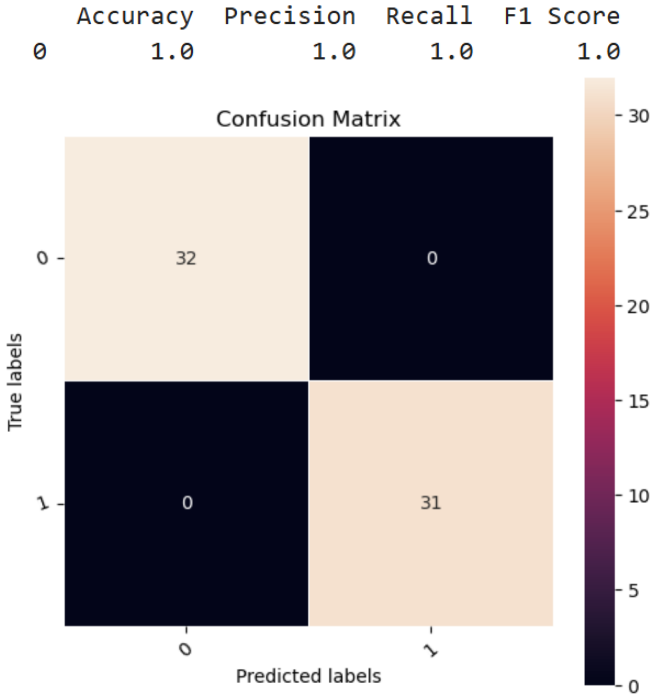


Figure 45: Model 4 Evaluation Metrics | Validation

- **Key Observations: -**
  - **Recall:**
    - Training: 100%
    - Validation: 100%
  - **Accuracy:**
    - Training: 100%
    - Validation: 100%
  - **False Negatives ('No Helmet' classified as 'Helmet')**
    - Training: 0 missed violations
    - Validation: 0 missed violation
  - **Generalization:** Validation performance is very consistent with training, indicating better generalization.
  - **To summarize:** -
    - Model shows **perfect performance**, with **no errors** on either training or validation sets.
    - Just like the previous model, deeper audit may be required, as it suggests **model memorizing the data due to small dataset.**
    - Due to small size of the dataset, it is easy to overfit. With only 630 images, especially if they have limited variability (same background, lighting, angles), a model can easily memorize features without learning to generalize. Also, if we are using 10% for validation, that's only 60 images. With such a small set, achieving 100% accuracy may just be luck or overfitting, not true generalization.
    - **In the final model (subsequent section), we would augment the data** as well to make the model more robust.

Visualizing Predictions

- Visualizing the predicted and correct label of 3 random images from validation data: -

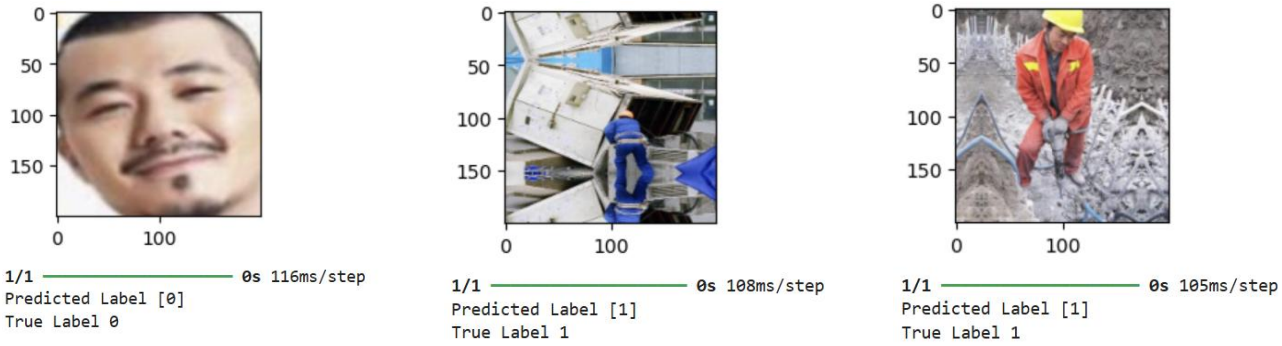


Figure 46: Model 4 – Visualizing Predictions

Model 5 – VGG-16 Model with FFNN & Data Augmentation using Original Dataset

Input Characteristics

- We will use the **Original Dataset** to build this model.
- **Transfer Learning** to be leveraged by loading a pre-built architecture, **VGG16**, which was trained on the ImageNet dataset and is the runner-up in the ImageNet competition in 2014.
- We will **directly use the convolutional and pooling layers and freeze their weights** i.e. no training will be done on them. We will remove the already-present fully-connected layers **and add our own fully-connected layers** for this binary classification task.
- For classification, we will add a Flatten layer and a Feed Forward Neural Network.
- We will also use a **Dropout layer** to reduce overfitting: -
  - Models tend to overfit when we have deep layers in the Neural Network.
  - Prevents overfitting by ensuring the model doesn't rely too heavily on any one feature or neuron.
- Below are the Input Image Characteristics from the Training Dataset of Images that would be fed as an input to the CNN Model: -

```
Model 5 Image Input Characteristics:

No. of Classes: 2
Image Size: 150528
Image Input Shape: (224, 224, 3)
(504, 224, 224, 3)
```

Figure 47: Model 5 Input Image Characteristics

  - Each Image is of size (224 x 224) with 3 channels (RGB)
  - Total Image Size = 224*224*3 = 1,50,528
  - 2 Label Classes: 0,1

Model Architecture

- **Loading VGG16 Model: -**

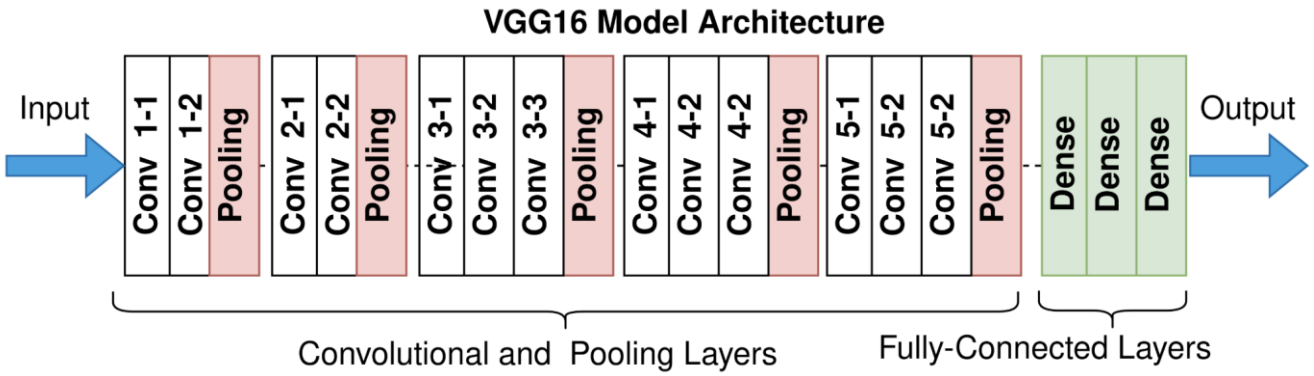


Figure 48: VGG16 Model Architecture

- VGG16 Model, pre-trained model with its weights, is loaded. Layer-types remain the same as described earlier, except, VGG16 has a peculiar architecture as shown above.
- To remove the Fully-connected layers of the imported pre-trained model, we can also specify an additional keyword argument – 'include_top = False'. Then, the model will be imported without the fully-connected layers.
- We won't have to do all the steps of getting the last convolutional layer and creating a separate model.
- Once, the **VGG16 Model (without Fully-connected Layers) is loaded with its weights, we freeze all the layers for training.**

- Below is the **Model Summary** of the **VGG16 Model (without Fully-connected Layers): -**

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0

Total params: 14,714,688 (56.13 MB)

Trainable params: 14,714,688 (56.13 MB)

Non-trainable params: 0 (0.00 B)

Figure 49: VGG16 Model Summary (without Fully-connected Layers)

```
input_layer_1 False
block1_conv1 False
block1_conv2 False
block1_pool False
block2_conv1 False
block2_conv2 False
block2_pool False
block3_conv1 False
block3_conv2 False
block3_conv3 False
block3_pool False
block4_conv1 False
block4_conv2 False
block4_conv3 False
block4_pool False
block5_conv1 False
block5_conv2 False
block5_conv3 False
block5_pool False
```

Figure 50: VGG16 Model (without Fully-connected Layers) – Layer Freeze for Training

- **Final Model Layers: -**
  - **VGG16 Model Layer:** This is the pre-trained convolutional part of VGG16, which extracts high-level image features like edges, textures, and object parts. It doesn't include the dense layers from VGG16's classifier.
    - VGG16 Convolution Layers (without fully-connected layers) with the Input Layer, which is already fed into VGG16 Model.
  - **Flatten Layer**: Converts the 3D image to a 1D vector i.e. (7,7,512) → (25088)
  - **Dense Layer 1 (Hidden Layer)**: Adds non-linearity, helps learn spatial features & learns complex patterns using neurons and 'relu' activation.
    - 256 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Dense Layer 2 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 128 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Dropout Layer:** Randomly deactivates neurons during training.
    - 50% of neurons are turned off randomly in this layer during each update. **In previous models, we suspected overfitting; so we introduced this layer between the Dense Layers to Regularize the Model this time.**
  - **Dense Layer 3 (Hidden Layer)**: Useful for deeper representation, but increases complexity as it adds more learning capacity, if needed.
    - 64 neurons with 'relu' activation and 'he_uniform' to initialize weights
  - **Output Layer:** Outputs a probability score between 0 and 1**.**
    - 1 neuron with 'sigmoid' activation.
- Below is the summary of the model: -

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 256)	6,422,784
dense_1 (Dense)	(None, 128)	32,896
dropout (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 64)	8,256
dense_3 (Dense)	(None, 1)	65

Total params: 21,178,689 (80.79 MB)

Trainable params: 6,464,001 (24.66 MB)

Non-trainable params: 14,714,688 (56.13 MB)

Figure 51: Model 5 Summary

Compile & Fit Model

- We Compile the Model using the following hyperparameters: -
  - **Loss = binary_crossentropy**
    - Cross-entropy is the most commonly used loss function for classification problems. Cross-entropy measures the difference between the true label and the predicted probability.
    - 'binary_crossentropy' is used as the loss function because the model is doing binary classification and ends with a sigmoid output. It's mathematically tailored for this setup and leads to better performance.
  - **Metrics = Accuracy**
    - Accuracy gives a quick sense of overall model correctness and since the dataset is balanced, the results are not misleading (not biased to majority class)
  - **Optimizer = Adam**
    - Adam optimizer is highly recommended for most deep learning tasks, as it combines the benefits of two powerful optimizers: AdaGrad and RMSProp.
    - Adaptive Learning Rates: Adam adjusts the learning rate for each parameter individually, based on how frequently it's updated and helps learn fine-grained image features like helmet edges or head shape more efficiently.

- Leverage Momentum: Adam uses momentum to accelerate learning in relevant directions.
- Together, they make learning: Faster, more stable & less sensitive to noisy gradients.
- Default hyperparameters (learning_rate=0.001) generally perform well and usually there is no need for extensive manual tuning.
- Adam handles small datasets (like your 631 images) better than SGD, because it adapts faster and doesn't require huge batch sizes.

- **Epoch = 15**
  - In the previous model, we have seen that the model is converging early up to 10-12 epoch.
  - To be on the safe side, we use 15 epochs this time to strike the right balance.
- **Batch-size = 32**
  - 32 is a power-of-2, which aligns well with GPU memory and parallel processing, balancing Training speed & Stability of gradient updates.
  - With only 631 images, a batch size of 32 gives about 20 batches per epoch, which is a good granularity for updates. Smaller batches (like 16) slow training; larger ones (like 64+) might overshoot minima or require more memory.
  - Only applies when training on raw NumPy arrays, not generators. In our case, this will be ignored (refer below section)
- **class_weight = <class_weights_calculated_above>**
  - If one class has significantly more samples than the other, a model might learn to just predict the majority class to get high accuracy. To counter this problem, this parameter tells the model to increase the penalty for misclassifying minority class & reduce the penalty for misclassifying majority class. Hence, it helps the model pay more attention to the underrepresented class.
  - In our case, even though the class weights are nearly balanced, it's a good practice to use this hyperparameter when we are using accuracy as one of key metrics.
- **Standardize Input Flow into the Model**
  - **Converts your NumPy arrays into a batch-generating iterator that feeds data into the model efficiently.**
  - Handles **Shuffling, Batching, and Seeding**: -
    I. **batch_size=128**
    II. **shuffle=True**
    III. **seed=42**
  - ImageDataGenerator.flow() Function is used for the same.
- **Data Augmentation**
  - ImageDataGenerator() is used to **artificially increase the diversity** of the training dataset by **applying random transformations** to the images **on the fly during training**.
  - Following parameters are defined for this data augmentation & transformation: -
    I. **rotation_range=20** | Randomly **rotates** images between **–20° to +20°**
    II. **width_shift_range=0.2 |** Shifts images **horizontally** by up to **20% of width**
    III. **height_shift_range=0.2 |** Shifts images **vertically** by up to **20% of height**
    IV. **shear_range=0.3 |** shears images **(slanting diagonally)** up to **±0.3 radians**
    V. **zoom_range=0.4 |** Randomly **zooms** in/out up to **40%**
    VI. **fill_mode='nearest' |** Fills in any **new empty pixels (from shifts/rotations)** using the **nearest pixel value**

- Below is the Model output, post Fit: -

```
Epoch 1/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 136s 34s/step - accuracy: 0.5191 - loss: 0.9478 - val_accuracy: 0.6984 - val_loss: 0.7171
Epoch 2/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 130s 34s/step - accuracy: 0.8543 - loss: 0.3537 - val_accuracy: 1.0000 - val_loss: 0.0028
Epoch 3/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 142s 33s/step - accuracy: 0.9519 - loss: 0.1449 - val_accuracy: 0.9841 - val_loss: 0.0383
Epoch 4/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 129s 33s/step - accuracy: 0.9855 - loss: 0.0525 - val_accuracy: 1.0000 - val_loss: 0.0016
Epoch 5/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 130s 33s/step - accuracy: 0.9887 - loss: 0.0498 - val_accuracy: 1.0000 - val_loss: 1.1661e-05
Epoch 6/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 130s 33s/step - accuracy: 0.9971 - loss: 0.0104 - val_accuracy: 1.0000 - val_loss: 2.9541e-06
Epoch 7/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 142s 34s/step - accuracy: 0.9798 - loss: 0.0359 - val_accuracy: 1.0000 - val_loss: 4.2830e-08
Epoch 8/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 130s 33s/step - accuracy: 0.9906 - loss: 0.0234 - val_accuracy: 1.0000 - val_loss: 5.0935e-08
Epoch 9/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 129s 33s/step - accuracy: 0.9926 - loss: 0.0383 - val_accuracy: 1.0000 - val_loss: 2.2928e-06
Epoch 10/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 130s 33s/step - accuracy: 0.9947 - loss: 0.0241 - val_accuracy: 1.0000 - val_loss: 4.8743e-07
Epoch 11/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 141s 33s/step - accuracy: 0.9965 - loss: 0.0164 - val_accuracy: 1.0000 - val_loss: 7.6112e-08
Epoch 12/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 129s 34s/step - accuracy: 0.9936 - loss: 0.0201 - val_accuracy: 1.0000 - val_loss: 4.4261e-07
Epoch 13/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 129s 34s/step - accuracy: 0.9979 - loss: 0.0112 - val_accuracy: 1.0000 - val_loss: 2.2240e-06
Epoch 14/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 131s 34s/step - accuracy: 0.9929 - loss: 0.0241 - val_accuracy: 1.0000 - val_loss: 1.4303e-05
Epoch 15/15
4/4 ━━━━━━━━━━━━━━━━━━━━ 130s 33s/step - accuracy: 0.9921 - loss: 0.0143 - val_accuracy: 1.0000 - val_loss: 2.0882e-06
```

Figure 52: Model 5 Output

Plot – Model Accuracy vs. Epoch

- Below is plot between Model Accuracy & Epoch for both Training & Validation Dataset: -

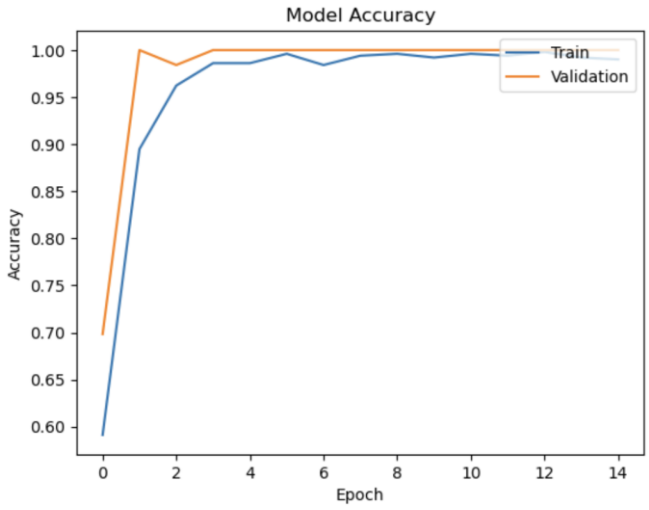


Figure 53: Model 5 | Model Accuracy Plot

- **Key Observations**: -
  - Model learns rapidly in the first few epochs (for both Train & Validation Accuracy), suggesting good feature learning.
  - Training Accuracy rises quickly 99-100%, indicating Model fits training data extremely well, possibly memorizing it.
  - Validation accuracy stays consistently high at 99-100%, indicating strong generalization with unseen data (post data augmentation).
  - No gaps observed between training and validation curves, suggesting no overfitting.
  - Both curves converge and stabilize at 99–100% by 6-8 epoch.

Model Evaluation

- Below are the Performance Metrics & Confusion Matrices for both Train & Validation Datasets: -

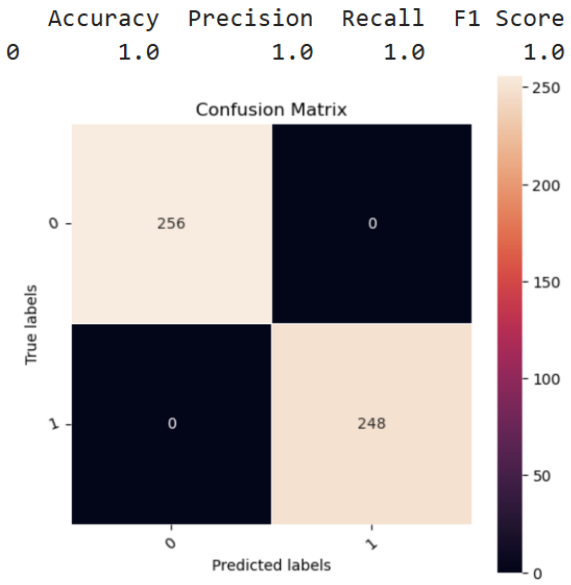


Figure 54: Model 5 Evaluation Metrics | Train

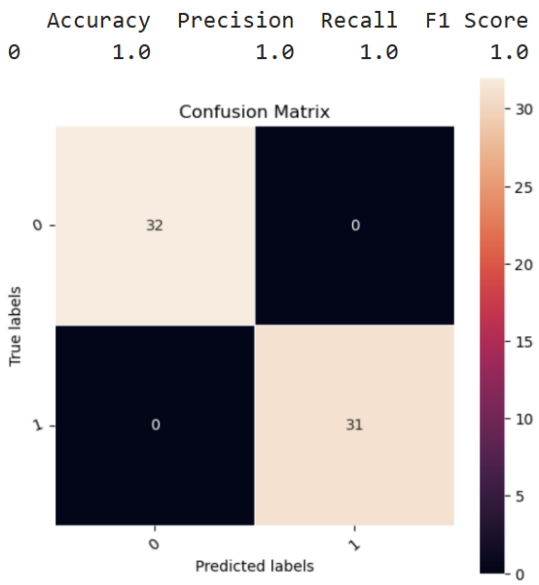


Figure 55: Model 5 Evaluation Metrics | Validation

- **Key Observations: -**
  - **Recall:**
    - Training: 100%
    - Validation: 100%
  - **Accuracy:**
    - Training: 100%
    - Validation: 100%
  - **False Negatives ('No Helmet' classified as 'Helmet')**
    - Training: 0 missed violations
    - Validation: 0 missed violation
  - **Generalization:** Validation performance is very consistent with training, indicating better generalization.
  - **To summarize:** -
    - Model shows **perfect performance**, with **no errors** on either training or validation sets.
    - Unlike previous models, this time we **augmented our data** & the **model still performed very well**, thus, enhancing generalization.
    - Even though, **we augmented the data**, there is still a **possibility** that the model is **memorizing** the data, given the dataset size constraint.
    - It is **recommended**, to **revalidate the model with a bigger dataset**.

Visualizing Predictions

- Visualizing the predicted and correct label of 3 random images from validation data: -

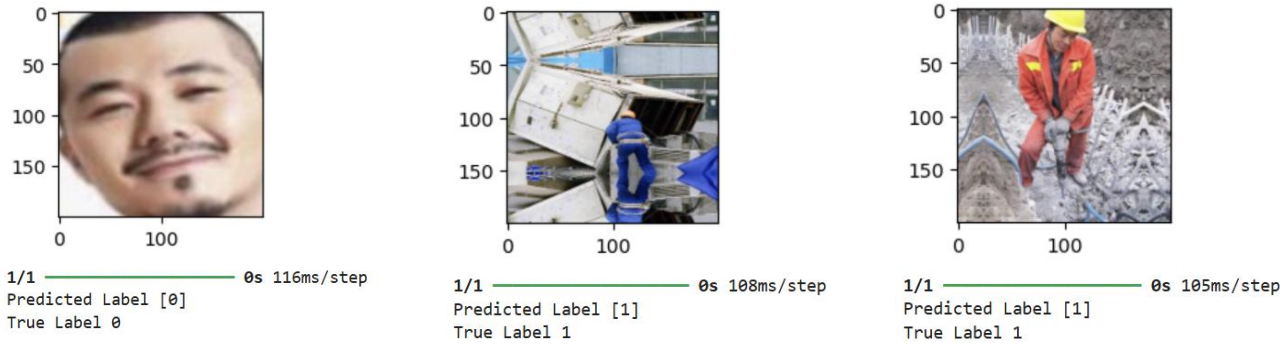


Figure 56: Model 5 – Visualizing Predictions

Rubric Question 4: Model Performance Comparison and Final Model Selection

Training Dataset Performance Comparison

- Below is the summary of performance comparison of Training dataset

	Simple ANN with Image Flattening	Simple ANN with Preprocessed Images (Grayscale) and Image Flattening	Basic Convolutional Neural Network (CNN) - Original Images)	VGG-16 Model with Feed Forward Neural Network (FFNN) - Original Images	VGG-16 Model with FFNN and Data Augmentation - Original Images
Accuracy	0.998016	0.950397	1.0	1.0	1.0
Precision	0.998024	0.952505	1.0	1.0	1.0
Recall	0.998016	0.950397	1.0	1.0	1.0
F1 Score	0.998016	0.950367	1.0	1.0	1.0

Figure 57: Training Dataset Performance Comparison

Validation Dataset Performance Comparison

- Below is the summary of performance comparison of Validation dataset

	Simple ANN with Image Flattening	Simple ANN with Preprocessed Images (Grayscale) and Image Flattening	Basic Convolutional Neural Network (CNN) - Original Images)	VGG-16 Model with Feed Forward Neural Network (FFNN) - Original Images	VGG-16 Model with FFNN and Data Augmentation - Original Images
Accuracy	0.888889	0.952381	1.0	1.0	1.0
Precision	0.908832	0.952861	1.0	1.0	1.0
Recall	0.888889	0.952381	1.0	1.0	1.0
F1 Score	0.887295	0.952381	1.0	1.0	1.0

Figure 58: Validation Dataset Performance Comparison

Training & Validation Dataset Performance Difference

- Below is the summary of performance difference between Training & Validation dataset

	Simple ANN with Image Flattening	Simple ANN with Preprocessed Images (Grayscale) and Image Flattening	Basic Convolutional Neural Network (CNN) - Original Images)	VGG-16 Model with Feed Forward Neural Network (FFNN) - Original Images	VGG-16 Model with FFNN and Data Augmentation - Original Images
Accuracy	0.109127	-0.001984	0.0	0.0	0.0
Precision	0.089192	-0.000356	0.0	0.0	0.0
Recall	0.109127	-0.001984	0.0	0.0	0.0
F1 Score	0.110720	-0.002014	0.0	0.0	0.0

Figure 59: Performance Difference between Training & Validation Datasets

Final Model Selection

- **Top Candidates: -**
  - **Model 5 | VGG-16 + FFNN + Data Augmentation**
    - Training & Validation Metrics (Recall-Focused): 100%
    - Best generalization despite exposure to unseen data through data augmentation
  - **Model 4 | VGG-16 + FFNN (No Data Augmentation)**
    - Training & Validation Metrics (Recall-Focused): 100%
    - High performance but may overfit due to no augmentation (considering, small dataset)

- **Rejected Models: -**
  - **Model 1 | Simple ANN model (Original Images)**
    - Training Metric (Recall): 99.80%
    - Validation Metric (Recall): 88.88%
    - Lower validation scores
    - Big Gap in Training vs Validation performance – possible overfitting.
    - No convolutional power for spatial feature extraction

- **Model 2 | Simple ANN model (Grayscale Images)**
  - Training Metric (Recall): 95.04%
  - Validation Metric (Recall): 95.24%
  - Relatively lower validation scores than the best model, even though, it offers better generalization than the 1st Model
  - No convolutional power for spatial feature extraction
- **Model 3 | Basic CNN (no Transfer Learning)**
  - Training & Validation Metrics (Recall-Focused): 100%
  - Relatively shallow in comparison to transfer learning models (in comparison to Deep, pretrained VGG-16 backbone)
  - Limited Generalization due to risk of overfitting on small dataset, as against Transfer Learning Models that enable robust feature extraction.
  - CNN can easily memorize small dataset, as against, data-augmented VGG16 that increases robustness & variation.
  - Training History-plot shows fluctuations and instability in validation accuracy

- **Final Model: -**
  - **Model 5 | VGG-16 + FFNN + Data Augmentation**
    - While the performance metrics are as perfect as Model 4, we choose Model 5 as **data augmentation enhances model generalization**.
    - Dataset has only 631 images, which is small for deep learning. **Without augmentation**, the model: -
      - ✓ Might **memorize training data** (even if validation still looks good now)
      - ✓ Might **fail on real images** taken from different angles, lighting, positions
    - However, **with augmentation**, the model: -
      - ✓ Learns on a slightly bigger dataset, **reducing the risk of memorizing** training data
      - ✓ Learns images that are **rotated, zoomed, sheared or shifted**
      - ✓ Learns more **diverse examples during training**
      - ✓ Learns **invariance to distortions** it may face in live CCTV, site cameras, etc.
    - Hence, **Model 5 is relatively the best** model because: -
      - ✓ **It simulates real-world noise**
      - ✓ **Improves generalization**
      - ✓ **Minimizes overfitting risk**
      - ✓ **Is production-ready for deployment in diverse industrial environments**

- After evaluating all models across both training and validation sets, the **VGG-16 model with Feedforward Neural Network (FFNN) and Data Augmentation is selected as the final model for further testing and deployment**. While both the plain VGG-16 + FFNN model and the augmented version achieved perfect scores (100%) on all performance metrics, the latter is clearly more robust and generalizable for real-world application. This superiority is due to the **use of data augmentation**, which **introduces controlled variability** in the training process (such as image rotation, shear, zoom, and shifts). These transformations help the model learn to detect helmets under varied conditions that **closely simulate actual worksite environments**, including different camera angles, lighting conditions, and worker postures. As a result, this model is **less likely to overfit** and is better equipped to **maintain high performance on unseen images**, making it the most reliable and production-ready choice for SafeGuard Corp's helmet detection system.

Test Dataset Performance of the Final Model

- Below are the Performance Metrics & Confusion Matrices for both Train & Test Datasets: -

	Accuracy	Precision	Recall	F1 Score
0	1.0	1.0	1.0	1.0

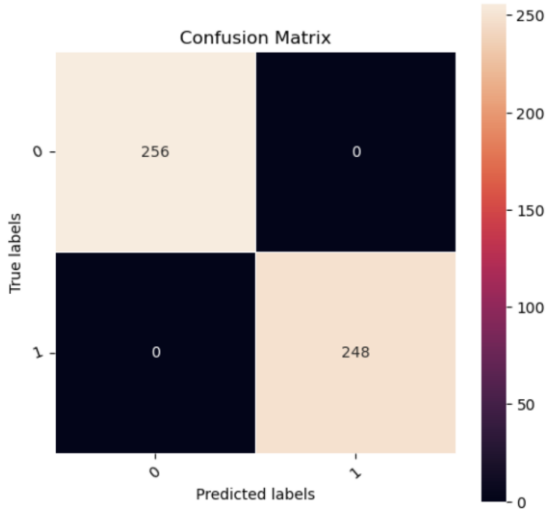


Figure 60: Final Model Evaluation Metrics | Train

	Accuracy	Precision	Recall	F1 Score
0	1.0	1.0	1.0	1.0

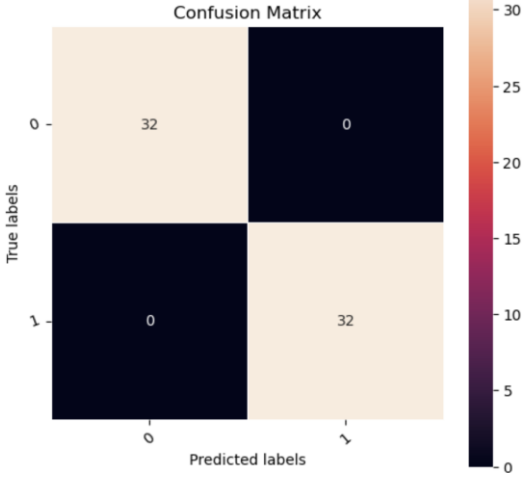


Figure 61: Final Model Evaluation Metrics | Test

- **Key Observations: -**
  - **Recall:**
    - Training: 100%
    - Validation: 100%
  - **Accuracy:**
    - Training: 100%
    - Validation: 100%
  - **False Negatives ('No Helmet' classified as 'Helmet')**
    - Training: 0 missed violations
    - Validation: 0 missed violation
  - **Generalization: Test performance is very consistent** with training, indicating better generalization.
  - **To summarize:** -
    - Model shows **perfect performance**, with **no errors** on either training or test sets.
    - Despite **data augmentation,** the **model still performed very well**, thus, enhancing generalization.
    - Even though, we augmented the data, there is still a **possibility** that the model is **memorizing** the data, given the dataset size constraint.
    - It is recommended, to **revalidate the model with a bigger dataset**.

Rubric Question 5: Actionable Insights & Recommendations

- **Actional Insights**
  1. **Transfer Learning Maximizes ROI on Limited Data:**
     - ✓ Leveraging a pretrained model like VGG-16 allowed us to achieve enterprise-grade accuracy without requiring thousands of labelled images.
     - ✓ This reduces data acquisition costs and speeds up deployment.

  2. **Data Augmentation Enhances Field Reliability**
     - ✓ Models trained with augmented images are better equipped to handle real-world conditions (e.g., different angles, lighting, and movement), ensuring more consistent safety compliance monitoring in live environments.

  3. **Simple Models Fall Short on Complex Visual Tasks**
     - ✓ Traditional neural networks failed to detect helmet violations accurately, highlighting the need for more sophisticated deep learning architectures in computer vision-based safety systems. Hence, the need to leverage models like VGG-16 using Transfer Learning.

  4. **Computer Vision Enables Scalable, Automated Safety Enforcement**
     - ✓ An AI-based helmet detection system minimizes reliance on manual supervision, reducing human error and operational overhead. This positions the solution as a cost-effective, scalable safety compliance mechanism across multiple sites and facilities.

- **Business Recommendations**
  1. **Integrate the Model into a Real-Time Monitoring System**
     - ✓ Deploy the selected model into CCTV or camera surveillance systems to enable real-time helmet compliance detection.
     - ✓ Automate alerts to site supervisors when violations are detected.
     - ✓ This will reduce manual oversight costs, enhance enforcement, and create audit trails for safety compliance.

  2. **Pilot the Solution Across Multiple Site Conditions**
     - ✓ Start with pilot deployments in varied environments (indoor, outdoor, day/night, factory/construction) to test the model's robustness under real conditions.
     - ✓ Collect performance feedback to identify blind spots (e.g., helmet color blending with background).

  3. **Establish a Feedback Loop for Continuous Learning**
     - ✓ Integrate a mechanism to capture misclassified images from live deployments.
     - ✓ Use these images to retrain the model periodically, improving accuracy over time and adapting to new safety gear or worksite setups.

  4. **Scale with a Safety Dashboard and Reporting System**
     - ✓ Develop a central dashboard that shows real-time compliance rates, flagged violations, and historical trends.
     - ✓ This helps safety officers track site-level risk, ensure regulatory compliance, and drive data-driven interventions.

  5. **Invest in a Broader PPE Detection Ecosystem**
     - ✓ Expand the system to detect additional PPE (e.g., vests, gloves, goggles) using the same vision-based framework.
     - ✓ Position SafeGuard Corp as a leader in AI-driven occupational safety solutions, not just helmet compliance.