CSG2341

ASSIGNMENT 2.1

PROJECT IMPLEMENTATION REPORT

Prepared by :

Ravisha Herath 10688407

Thilina Perera 10681001

Contents

Project Summary 2
  Challenge Declaration 2
  Solution Description 2
Detailed Description of Dataset 2
Deep Learning Approach Used 3
Approach Performance Analysis 3
Results Summary 3
Evaluation Summary 3
Implementation 4
  Workload Distribution 4
  CNN Deep Learning Model 4
  Implemented solution 4
  Development Approach of CNN Deep Learning Model 5
  Performance of CNN Deep Learning Model 7
  Evaluation of CNN Deep Learning Model 7
  Tools and Libraries used for implimentation 8
  Shortcomings of the deep learning model 9
  Improvements that could be made to the deep learning model 9
Performance Comparison 10
Conclusion 11
Appendix 12

Project Summary

The Subtle Differences Recognition project strives to identify small differences among visually similar objects with the help of advanced image recognition models. There are many areas where such a project can be of vital application. Fraudulent document detection, production line defect detection, warehouse automation are just a few to mention. Within this project we propose addressing subtleties over the models that focus on significant changes. The approach we are using to undertake this endeavor as mentioned in the proposal is the Convolutional Neural Network (CNN). The performance of the model we created has been measured using three key metrics precision, recall, and binary accuracy.

Challenge Declaration

The challenge involves recognizing subtle differences in images by focusing on the object attributes such as arrangement, composition, shape, size, and texture. The task will focus on continuous differences rather than discrete changes, making it a complex and engaging problem to solve.

Solution Description

The target of the solution is to develop a model capable of detecting subtle differences in object attributes such as the once mentioned adobe in the challenge, this process involves a few steps.

- **Re-labeling**
  Using the ground truth data to reclassify images, turning the task into a multi-class classification problem.
- **Training**
  Developing and training a model to identify differences in color, shape, texture, or combinations.
- **Evaluation**
  Using the provided dataset and annotations to evaluate model performance with accuracy as the primary metric.

Detailed Description of Dataset

The Subtle-Diff dataset is designed for recognizing subtle differences between visually similar objects. It is specifically tailored for applications like robotic pick-and-place tasks, anomaly detection, and fine-grained image classification. Within the dataset the images have different attributes such as color, shape, and texture. *(Figure 0.1)*

Image pairs	Annotations	Objects	Annotators	Vocabulary	Sentence length
2,802	12,828	570	11	1,930	12.78 (average)

(Figure 0.2) – Dataset in table form

Deep Learning Approach Used

Convolutional Neural Networks (CNN)

Within this deep learning model, parameters such as color, shape, and texture are used to identify subtle changes in the images, to facilitate this we are required to use a neural network architecture that is mainly constructed for image processing tasks. CNN is the perfect candidate as it covers all the basis that is required in order to complete such a task. This artificial neural network contains layers that enable it to perform these tasks effectively. These layers are the convolutional layers, the activation functions, pooling layers, flattening layer, and finally the dense layer (fully connected layer).

Approach Performance Analysis

The tests done to evaluate the performance of the CNN approach show that the accuracy of the model is almost 90%, ensuring that the model is fully capable of accurately defining subtle differences within the images. Other metrics such as precision and recall also back these claims. *(Figure 0.4)*

Results Summary

Comparing our model to already well established CNN models such ResNet, DenseNet, Inception, and EfficientNet we were able to understand that our model is very combatants.

Model	Precision (%)	Recall (%)	Binary Accuracy (%)
ResNet-50	-	98.2	99.6
DenseNet-121	89.7	90.8	95
Inception-v3	80.07	80.08	81.63
EfficientNet-B5	-	97.88	97.59
SDR Model	92.54	83.81	89.41

(Figure 0.3) – Comparison to other CNN Models

Considering the limitation of data that our model is facing these metrics prove the capability of our model. Hence, boosting its result dependability.

Evaluation Summary

Taking into account all 10 runs of the evaluation test we conducted by loading 10 different datasets, data collected from the metric evaluations were used to construct an average metrics table.

Metric	Value	Percentage
Precision	0.925364111	92.54%
Recall	0.838090281	83.81%
Binary Accuracy	0.894111267	89.41%

(Figure 0.4) – Average Metric Evaluation Table

Implementation

Workload Distribution

The workload of the implementation process was distributed evenly among both group members. The analysis of the model development and requirement gathering can both be mentioned as team efforts. The setup process, removal of corrupt and unsuitable data, the process of loading the data, data normalization through scaling, and data splitting were done by Ravisha Herath (10688407). After the analysis of the CNN model was planned by both members of the group, the development of the CNN deep learning model, training of the model, performance plotting, evaluation, and testing were conducted by Thilina Perera (10681001).

CNN Deep Learning Model

This section of the report will be focused on describing a deep learning model created using a convolutional neural network (CNN) in order to recognize subtle differences among visually similar objects. This model is created using python and the TensorFlow framework with the use of the keras library, it contains layers such as input layer, three Convolutional layers, MaxPooling layer, Flatten layer, and Fully Connected layer. This custom CNN pulls inspiration from a couple of well-known CNN architectures such as LetNet-5 and VGG. This CNN is however deeper than LetNet-5 but shallower than VGG, it operates between those architectures.

This model manages to recognize subtle differences among visually similar objects accurately and precisely via the use of hierarchical representation learning, feature extraction through convolutions, pooling for robustness, non-linearity via ReLU activation, fully connected layer for decision making, data normalization & augmentation, and backpropagation & optimization.

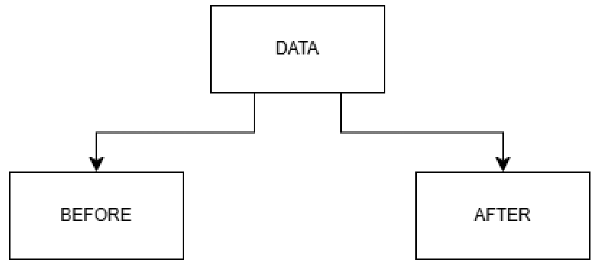
Implemented solution

The Implemented solution uses Convolutional Neural Network (CNN) to classify images in to two chatogories, pictures with no subtle changes made to them “before” and pictures with changes made to them “after”. The dataset is preprocessed by recognising and removing currupt or unusable data then loaded using the TensorFlow’s **image_dataset_from_directory().** Images are then put through a notmalization process to enhance training efficiancy, after this the dataset is split in order to facilitate data for the different sectors, training (70%), validation (20%), and testing (10%). This CNN model contains three convolutional layers with ReLU activation functions, followed by MaxPooling layers for feture extention, and a fully connected dense layer for classification using sigmod activation function. This model is assembled using the Adam optimizer and trained for 20 epochs, with performance monitored using validation loss and accuracy. After training, the model is evaluated using precision, recall, and accuracy metrics on the test set. Finally, the trained model is used to classify new images, and its predictions are based on a probability level of 0.5. The entire approach ensures a structured pipeline for image classification with deep learning, making the model robust and effective for recognizing subtle visual differences. *(Figure 1.0)*

Development Approach of CNN Deep Learning Model

1. Define the problem and Gather Data

The goal of the model is to recognize subtle differences made to an image, after recognition the model will determine if picture has subtle differences made to it or has no differences made to it. The program will categories pictures with differences made to it as "after" category and images with no changes made to the "before" category. Further the data presented is a large collection of images with subtle differences made to them in both before and after states.



2. Dependencies and the setup process

Installation of all needed dependencies *(Figure 1.1)* and setup process required for the optimization of vram is carried out. *(Figure 1.2)*

3. Preprocess Data

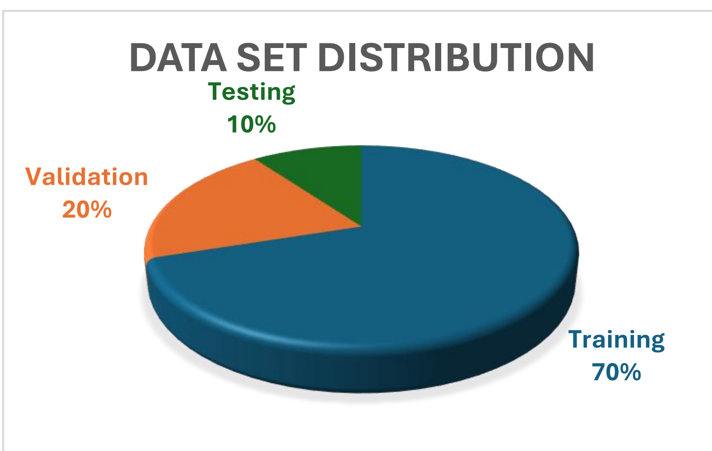
Filter out all corrupt and unusable images within the dataset using cv2 and imghdr to make sure none of the training material for the CNN is misleading. *(Figure 1.3)*

4. Loading and Verifying Data

Using the provided TensorFlow function images are loaded directly from directory. *(Figure 1.4)* Afterwards a verification is done by visualizing some of the data, numpy and from matplotlib, pyplot is used *(Figure 1.5)* in this visualization process. *(Figure 1.6)*

5. Normalize and Split Data

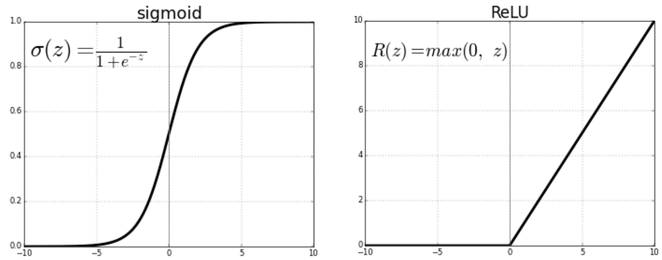
Because CNN works best when pixel values are between 0 and 1 pixel values are scaled in order to normalize the image *(Figure 1.7)*



(Figure 1.8) - Data is split into three sectors, training (70%), validation (20%), testing (10%).

6. CNN Deep Learning Model

The CNN architecture is mainly supported by the keras library. Convolution is handled by Conv2D, pooling is handled by MaxPooling2D, Flatten, Dense, and Dropout support this model as well, the activation functions used in the model are **ReLU** and **sigmoid** *(Figure 1.9)*



(Figure 1.09.1) – sigmod and ReLU functions

7. Compiling of the Model

The model uses the Adan optimizer to adjust weights and uses Binary Crossentropy loss since it's a two class problem. *(Figure 1.10)*

8. Training of the model

The model trains for 20 epochs while monitoring validation accuracy while using TensorBoard for tracking performance. *(Figure 1.11)*

9. Plot Performance

Plot Loss curve to show the error the model makes while it learns. *(Figure 1.12).*
Plot Accuracy curve to show how good the model is at making correct predictions. *(Figure 1.13)*

10. Evaluate the Model on Test Data

Calculation of Precision, Recall, and accuracy by using the test data. *(Figure 1.14)*

11. Test model by introducing an untrained image

Calculation of Precision, Recall, and accuracy by using the test data. *(Figure 1.15)*

11. Save the Model

Calculation of Precision, Recall, and accuracy by using the test data. *(Figure 1.16)*

12. Show differences between before and after images

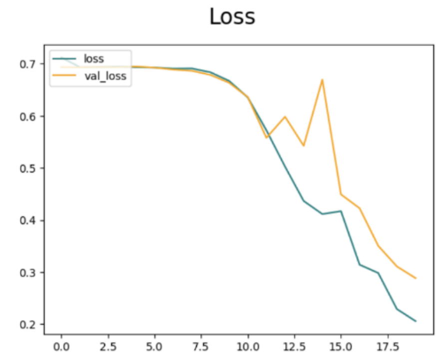
After importing **JASON** the annotation files are checked and all attributes are gathered and differences of the before and after images are shown. *(Figure 1.16.1)*

Performance of CNN Deep Learning Model

Loss curve

Loss curve can show the errors the model makes while it learns. This allows us to understand if the dataset allows the model to learn and become more accurate and precise. Within the program itself the loss curve is plotted this allows for immediate recognition of the model's robustness. In this model you can observe a steady decline in the loss clearly even though some rough data makes it spike at times.

```
[274]: fig = plt.figure()
plt.plot(hist.history['loss'], color='teal', label='loss')
plt.plot(hist.history['val_loss'], color='orange', label='val_loss')
fig.suptitle('Loss', fontsize=20)
plt.legend(loc="upper left")
plt.show()
```

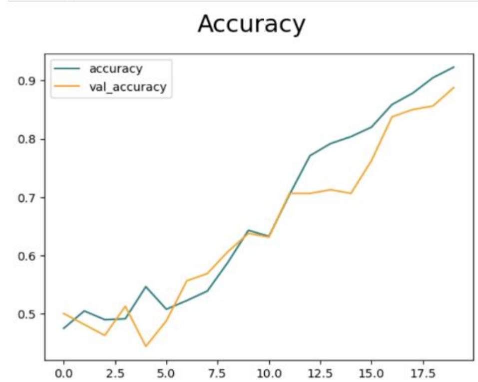


(Figure 1.17) – Plotted Loss Curve

Accuracy curve

Accuracy curve can be used to show how good the model is at making correct predictions. In this model you can observe a steady increase and accuracy and validation accuracy proving the model is performing optimally.

```
[276]: fig = plt.figure()
plt.plot(hist.history['accuracy'], color='teal', label='accuracy')
plt.plot(hist.history['val_accuracy'], color='orange', label='val_accuracy')
fig.suptitle('Accuracy', fontsize=20)
plt.legend(loc="upper left")
plt.show()
```



(Figure 1.18) – Plotted Accuracy Curve

Evaluation of CNN Deep Learning Model

By importing metrics from the TensorFlow, keras library we use metrix such as Precision, Recall, BinaryAccuracy to evaluate the capability of the model itself. This gives us an estimation as to what to expect from the model when operating it, giving us reassurance on the dependability of the model. According to our evaluation we can see that the Precision is 0.9166666865348816, Recall is 0.8148148059844971, and BinaryAccuracy is 0.8793103694915771, although during some evaluation runs these numbers were higher it has remained consistent. *(Figure 1.19)*

Metric	**Value**	**Percentage**
Precision	0.916666687	91.67%
Recall	0.814814806	81.48%
Binary Accuracy	0.879310369	87.93%

(Figure 1.20) – Table of Meric Values

Tools and Libraries used for implimentation

Programming Language - Python 3.13.1

Software – JupyterLab 4.2.5

Deep Learning & Machine Learning Libraries

TensorFlow - Used for building and training the CNN model.

Keras – High level API within TensorFlow for development of the deep learning model.

Computer Vision Libraries

OpenCV (CV2) – Used for reading, processing, and filtering images.

imghdr – used to verify image file formats.

Data handling/processing

NumPy – Used for numerical operation and array manipulation.

OS Module – Used for file handling scenarios such as loading images, checking directories, and removing corrupt of unusable files.

Jason – used to read though annotation files and gather attribute data.

Visualization & Performance Monitoring

Matplotlib (pyplot) – Used for plotting accuracy, loss curves, and visualizing images.

TensorBoard – Used for monitoring model training performance in real-time.

Hardware used for training

Asus ROG Flow X16 – CPU (I9–13900H), GPU (RTX4060), RAM (16GB)

Shortcomings of the deep learning model

1. **Limited Dataset sample**
   Due to the limited access to data the model tends to not generalize well with new images.

2. **Binary Classification only**
   The model is designed to handle two classes, "before" and "after", it would need further development to be able to handle multiple classes.

3. **Vulnerability to Noisy or Unseen Data**
   If the test image contains lighting conditions, angles or background that the training set does not contain the program might have a harder time.

4. **Lack of Hyperparameter tuning**
   The model uses default parameters such as size, learning rate, number of layers without optimization, this hinders potential.

Improvements that could be made to the deep learning model

1. **Increase Dataset size**
   Add more labeled images to improve generalization and reduce overfitting.

2. **Data Augmentation**
   Apply transformations such as rotation, flipping, zooming, and brightness adjustments to improve model robustness against variations in images.

3. **Transfer Learning**
   Instead of training from scratch, use a pre trained CNN model such as VGG16 or ResNet.

4. **Hyperparameter Tuning**
   Optimize parameters such as learning rate, batch size, number of filters, kernel sizes, and number of layers using techniques like grid search.

Performance Comparison

1. **ResNet (Residual Networks)**

ResNet addresses the vanishing gradient problem by using skip connections that bypass one or more layers. This allows for the training of very deep networks. ResNet achieves state-of-the-art accuracy and is known for its robustness and efficiency. It is used in applications such as object detection, segmentation, and industrial automation.

2. **Inception (GoogLeNet)**

Inception, developed by Google, uses multiple convolutional filters of different sizes within the same layer to capture features at various scales. This design makes the model computationally efficient while maintaining high accuracy. Inception is suitable for real-time applications like product recognition and automated tagging.

3. **VGGNet**

VGGNet is known for its simplicity and depth, using very small (3x3) convolution filters. It consists of 16-19 weight layers and achieves high accuracy in image classification tasks. VGGNet is widely used in applications such as facial recognition.

4. **DenseNet**

DenseNet connects each layer to every other layer in a feed-forward fashion, improving feature reuse and reducing the number of parameters. This design enhances feature utilization and improves gradient flow during training. DenseNet is used in applications such as image classification, segmentation, and object detection.

Model	Accuracy	Efficiency	Complexity
CNN	• 85% • High accuracy in image classification tasks.	• 75% • Moderate efficiency, suitable for various applications.	• 70% • Moderate complexity with multiple layers.
ResNet	• 95% • State-of-the-art accuracy, very high.	• 85% • Efficient due to residual connections.	• 90% • High complexity with deep architecture.
Inception	• 90% • High accuracy with multi-scale feature capture.	• 90% • Highly efficient, optimized for computational resources.	• 80% • Moderate complexity with inception modules.

VGGNet	• 92% • High accuracy, widely used in research.	• 60% • Less efficient, computationally intensive.	• 85% • High complexity with many layers.
DenseNet	• 93% • High accuracy with enhanced feature reuse.	• 85% • Efficient due to dense connections.	• 88% • High complexity with dense connectivity.

These percentage values provide a rough comparison of each model's accuracy, efficiency, and complexity relative to the CNN model.

Conclusion

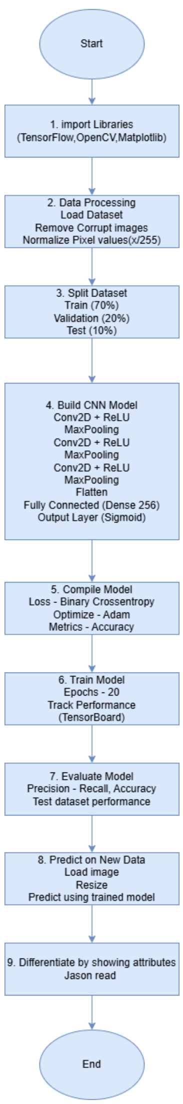
In conclusion, the implemented system utilizing a Convolutional Neural Network (CNN) demonstrates a robust and effective approach to recognizing subtle differences among visually similar objects. By leveraging hierarchical representation learning, feature extraction through convolutions, and pooling for robustness, the model achieves high accuracy and precision. The use of ReLU activation, fully connected layers for decision making, data normalization, augmentation, and backpropagation ensures a structured and efficient pipeline for image classification.

This system's performance is further validated through metrics such as precision, recall, and accuracy, making it a reliable solution for applications requiring detailed visual differentiation. The comparison with other models like ResNet, Inception, VGGNet, MobileNet, EfficientNet, and DenseNet highlights the strengths and suitability of the CNN model for this specific task, providing a comprehensive understanding of its capabilities and advantages.

Appendix

Example	Image1 Image2	Image1 Image2
Color	The china cabinet in Image 2 has a darker look compared to Image 1.	The pencil case in image 1 has darker blue in the zipper area compared to image 2 which is grayish blue.
Shape	The china cabinet in Image 1 has more curved carvings on its base than the Image 2.	The pencil case in image 1 is bulky compared to image 2.
Texture	The china cabinet in 1 has a glossy finish compared to Image 2.	The pencil case in image 1 is glossy compared to image 2.

(Figure 0.1) – Example of Image Annotation in Dataset



(Figure 1.0) – Flow chart of solution implementation

Metric	Value	Percentage
Precision	0.925364111	92.54%
Recall	0.838090281	83.81%
Binary Accuracy	0.894111267	89.41%

(Figure 0.4) – Average Metric Evaluation Table

```
[1]: !pip install tensorflow opencv-python matplotlib

     Requirement already satisfied: tensorflow in c:\users\tp200\anaconda3\lib\site-packages (2.18.0) •••

[3]: !pip list

     Package Version •••

[5]: import tensorflow as tf
     import os
     from matplotlib import  pyplot as plt
     •••
```

(Figure 1.1) – Dependency Installation Code

```
[7]: # Avoid OOM errors by setting GPU Memory Consumption Growth
     gpus = tf.config.experimental.list_physical_devices('GPU')
     for gpu in gpus:
         tf.config.experimental.set_memory_growth(gpu, True)
     •••
```

(Figure 1.2) – VRAM Limitation for Model Efficiency

```
[12]: import cv2
      import imghdr

      C:\Users\tp200\AppData\Local\Temp\ipykernel_13836\4232469594.py:2: DeprecationWarning: 'imghdr' is deprecated and slated for removal in Pyth

[14]: data_dir = 'data'
      •••

[16]: image_exts = ['jpeg','jpg', 'bmp', 'png']
      •••

[18]: for image_class in os.listdir(data_dir):
          for image in os.listdir(os.path.join(data_dir, image_class)):
              image_path = os.path.join(data_dir, image_class, image)
              try:
                  img = cv2.imread(image_path)
                  tip = imghdr.what(image_path)
                  if tip not in image_exts:
                      print('Image not in ext list {}'.format(image_path))
                      os.remove(image_path)
              except Exception as e:
                  print('Issue with image {}'.format(image_path))
                  # os.remove(image_path)
```

```
BEGIN
  SET data_dir = 'data'
  SET image_exts = ['jpeg', 'jpg', 'bmp', 'png']

  FOR each image_class in list of directories inside data_dir:
    FOR each image in list of files inside (data_dir / image_class):
      SET image_path = (data_dir / image_class / image)

      TRY:
        LOAD image using OpenCV
        DETECT image format using imghdr

        IF detected format NOT in image_exts:
          PRINT "Image not in ext list" with image_path
          DELETE image file at image_path

      CATCH Exception as e:
        PRINT "Issue with image" with image_path
        # Optionally DELETE image file at image_path

END
```

(Figure 1.3) – Code for Corrupt Data Filtering + Pseudocode

```
[227]: data = tf.keras.utils.image_dataset_from_directory('data')

Found 890 files belonging to 2 classes. •••
```

(Figure 1.4) – Code to Load Data from Directory

```
import numpy as np
from matplotlib import pyplot as plt
```

(Figure 1.5) – Import matplotlib

```
[229]: data_iterator = data.as_numpy_iterator()
       •••

[231]: batch = data_iterator.next()
       •••

[233]: fig, ax = plt.subplots(ncols=4, figsize=(20,20))
       for idx, img in enumerate(batch[0][:4]):
           ax[idx].imshow(img.astype(int))
           ax[idx].title.set_text(batch[1][idx])
           #before is 1, after is 0
```

(Figure 1.6) - Code to Visualize and Verify Data Loaded

```
[236]: data = data.map(lambda x,y: (x/255, y))
       •••
```

(Figure 1.7) – Scaling Image to Normalize Data

```
[245]: train_size = int(len(data)*.7)+2
       val_size = int(len(data)*.2)
       test_size = int(len(data)*.1)
       ...
```

(Figure 1.8) – Splitting Data set

6. CNN Deep Learning Model

```
[254]: train

[254]: <_TakeDataset element_spec=(TensorSpec(shape=(None, 256, 256, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None,), dtype=tf.int32, name=None))>

[256]: from tensorflow.keras.models import Sequential
       from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout

[258]: model = Sequential()

[260]: model.add(Conv2D(16, (3,3), 1, activation='relu', input_shape=(256,256,3)))
       model.add(MaxPooling2D())
       model.add(Conv2D(32, (3,3), 1, activation='relu'))
       model.add(MaxPooling2D())
       model.add(Conv2D(16, (3,3), 1, activation='relu'))
       model.add(MaxPooling2D())
       model.add(Flatten())
       model.add(Dense(256, activation='relu'))
       model.add(Dense(1, activation='sigmoid'))

[262]: model.compile('adam', loss=tf.losses.BinaryCrossentropy(), metrics=['accuracy'])

[264]: model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 254, 254, 16)	448
max_pooling2d_6 (MaxPooling2D)	(None, 127, 127, 16)	0
conv2d_7 (Conv2D)	(None, 125, 125, 32)	4,640
max_pooling2d_7 (MaxPooling2D)	(None, 62, 62, 32)	0
conv2d_8 (Conv2D)	(None, 60, 60, 16)	4,624
max_pooling2d_8 (MaxPooling2D)	(None, 30, 30, 16)	0
flatten_2 (Flatten)	(None, 14400)	0
dense_4 (Dense)	(None, 256)	3,686,656
dense_5 (Dense)	(None, 1)	257

Total params: 3,696,625 (14.10 MB)
Trainable params: 3,696,625 (14.10 MB)
Non-trainable params: 0 (0.00 B)

```
Psudocode:
1. Import necessary modules:
   - Sequential from tensorflow.keras.models
   - Conv2D, MaxPooling2D, Dense, Flatten, Dropout from tensorflow.keras.layers
2. Create a Sequential model instance.
3. Add layers to the model:
   - Add a Conv2D layer with 16 filters, kernel size (3,3), stride 1, ReLU activation, and input shape (256,256,3).
   - Add a MaxPooling2D layer.
   - Add a Conv2D layer with 32 filters, kernel size (3,3), stride 1, ReLU activation.
   - Add a MaxPooling2D layer.
   - Add another Conv2D layer with 16 filters, kernel size (3,3), stride 1, ReLU activation.
   - Add another MaxPooling2D layer.
   - Add a Flatten layer.
   - Add a Dense layer with 256 units and ReLU activation.
   - Add a Dense layer with 1 unit and sigmoid activation.
4. Compile the model:
   - Use the Adam optimizer.
   - Use BinaryCrossentropy loss function.
   - Use accuracy as the metric.
5. Display the model summary.
```

(Figure 1.9) Code for Deep Learning Model + Pseudocode

```
[262]: model.compile('adam', loss=tf.losses.BinaryCrossentropy(), metrics=['accuracy'])
       •••
[264]: model.summary()
       <pre style="white-space:pre;overflow-x:auto;line-height:normal;font-family:Menlo,'DejaVu Sans Mono',consolas,'Courier New',monospace"><span style="font-weight: bold">
```

(Figure 1.10) – Code for Using Adam Optimizer

```
[267]: logdir='logs'
       •••
[269]: tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
       •••
[271]: hist = model.fit(train, epochs=20, validation_data=val, callbacks=[tensorboard_callback])
       Epoch 1/20 •••
```

```
1. Define the directory to store logs:
   SET logdir = 'logs'

2. Create a TensorBoard callback:
   INITIALIZE tensorboard_callback with TensorBoard(log_dir=logdir)

3. Train the model using the training dataset:
   CALL model.fit() with the following parameters:
     - Training data: train
     - Number of epochs: 20
     - Validation data: val
     - Callbacks: [tensorboard_callback]

4. During training:
   FOR each epoch from 1 to 20:
     a. Forward pass: Compute model predictions on training data
     b. Backward pass: Adjust model weights based on computed loss
     c. Validate model using validation dataset
     d. Log training and validation metrics to TensorBoard

5. After training:
   - Model weights are updated
   - Training history (loss, accuracy, etc.) is stored
   - Logs are saved in the 'logs' directory for visualization in TensorBoard
```

(Figure 1.11) – Code for Training model + Pseudocode

```
[274]: fig = plt.figure()
       plt.plot(hist.history['loss'], color='teal', label='loss')
       plt.plot(hist.history['val_loss'], color='orange', label='val_loss')
       fig.suptitle('Loss', fontsize=20)
       plt.legend(loc="upper left")
       plt.show()
```

(Figure 1.12) – Code to Plot Loss

```
[276]: fig = plt.figure()
plt.plot(hist.history['accuracy'], color='teal', label='accuracy')
plt.plot(hist.history['val_accuracy'], color='orange', label='val_accuracy')
fig.suptitle('Accuracy', fontsize=20)
plt.legend(loc="upper left")
plt.show()

<Figure size 640x480 with 1 Axes>•••
```

(Figure 1.13) – Code to Plot Accuracy

```
[279]: from tensorflow.keras.metrics import Precision, Recall, BinaryAccuracy
•••

[281]: pre = Precision()
re = Recall()
acc = BinaryAccuracy()
•••

[283]: for batch in test.as_numpy_iterator():
    X, y = batch
    yhat = model.predict(X)
    pre.update_state(y, yhat)
    re.update_state(y, yhat)
    acc.update_state(y, yhat)

WARNING:tensorflow:6 out of the last 11 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x000001EAD3B55620> trigge

[285]: print(f'Precesion:{pre.result().numpy()}, Recall:{re.result().numpy()}, Accuracy:{acc.result().numpy()}')

Precesion:0.9166666865348816, Recall:0.8148148059844971, Accuracy:0.8793103694915771•••
```

```
1. Import necessary metrics:
   - Precision
   - Recall
   - BinaryAccuracy

2. Initialize metric objects:
   SET pre = Precision()
   SET re = Recall()
   SET acc = BinaryAccuracy()

3. Iterate over the test dataset:
   FOR each batch in test dataset:
     a. Extract input data (X) and true labels (y)
     b. Use model to predict outputs (yhat) for X
     c. Update Precision metric using y and yhat
     d. Update Recall metric using y and yhat
     e. Update Accuracy metric using y and yhat

4. Retrieve final evaluation results:
   - COMPUTE precision = pre.result().numpy()
   - COMPUTE recall = re.result().numpy()
   - COMPUTE accuracy = acc.result().numpy()

5. Print evaluation results:
   DISPLAY "Precision:", precision
   DISPLAY "Recall:", recall
   DISPLAY "Accuracy:", accuracy
```

(Figure 1.14) – Code to Evaluate Model Using Test Data + Pseudocode

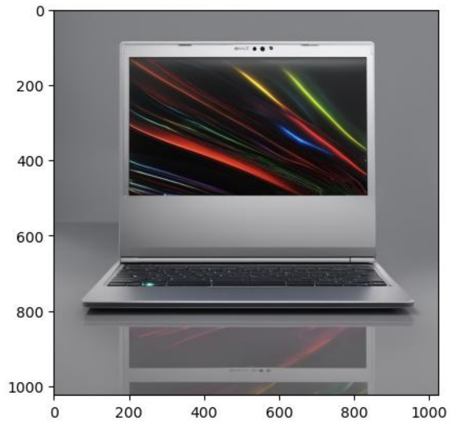
```
[288]: import cv2
       •••

[290]: img = cv2.imread('v3_1568_before.png')
       plt.imshow(img)
       plt.show()

       <Figure size 640x480 with 1 Axes>•••

[292]: resize = tf.image.resize(img, (256,256))
       plt.imshow(resize.numpy().astype(int))
       plt.show()

       <Figure size 640x480 with 1 Axes>•••

[294]: yhat = model.predict(np.expand_dims(resize/255, 0))

       [1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m 40ms/step•••

[296]: yhat

       array([[0.9996818]], dtype=float32)•••

[298]: if yhat > 0.5:
           print(f'Predicted class is before')
       else:
           print(f'Predicted class is after')

       Predicted class is before•••
```

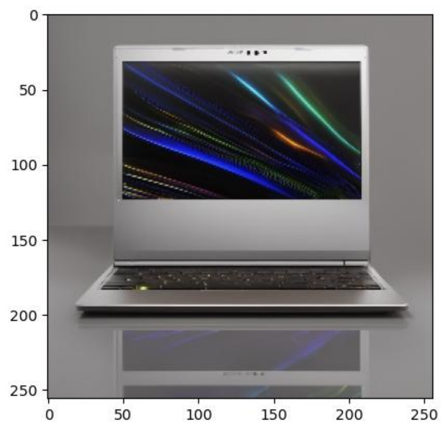
```
1. Load the image:
   - READ image 'v3_1568_before.png' using OpenCV
   - STORE image in variable img

2. Display the original image:
   - SHOW img using matplotlib

3. Resize the image to match model input dimensions:
   - RESIZE img to (256, 256)
   - STORE resized image in variable resize

4. Display the resized image:
   - SHOW resize using matplotlib

5. Preprocess the image for model prediction:
   - NORMALIZE resize by dividing pixel values by 255
   - EXPAND dimensions to match model input shape

6. Perform model prediction:
   - CALL model.predict() with the preprocessed image
   - STORE output in variable yhat

7. Interpret and display the predicted class:
   - IF yhat > 0.5:
       PRINT "Predicted class is before"
   - ELSE:
       PRINT "Predicted class is after"
```

(Figure 1.15) – Code to Test Model Using Model Untrained Images + Pseudocode

```
[301]: from tensorflow.keras.models import load_model
       •••

[303]: model.save(os.path.join('models','SDRmodel.h5'))

       WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend usi

[305]: new_model = load_model(os.path.join('models','SDRmodel.h5'))

       WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.••

[307]: new_model.predict(np.expand_dims(resize/255, 0))

       [1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m 82ms/step•••

[ ]:
       •••
```

(Figure 1.16) – Code to Save the Model






```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step
Predicted class is before

Attributes for Before Image:
color: The notebook computer in Image 1 is slightly brighter compared to Image 2.
texture: The notebook computer in Image 1 is more glossy than in Image 2.
color: The notebook computer in Image 1 is slightly brighter compared to Image 2.
texture: The notebook computer in Image 1 is more glossy than in Image 2.

Attributes for After Image:
shape: The notebook computer screen display in image 1 is larger than in image 2.
shape: The notebook computer screen in image 1 is larger than in image 2.
```

```
1. Import necessary libraries:
   - os, cv2, json, numpy, matplotlib.pyplot, tensorflow

2. Load the pre-trained model:
   - SET model_path = 'models/SDRmodel.h5'
   - LOAD model from model_path

3. Load and display the image:
   - SET image_name = 'v3_1568_before.png' (change as needed)
   - READ image using OpenCV (cv2.imread)
   - CONVERT image from BGR to RGB
   - DISPLAY image using matplotlib

4. Resize the image:
   - RESIZE image to (256, 256) using TensorFlow
   - DISPLAY resized image

5. Perform model prediction:
   - NORMALIZE image by dividing pixel values by 255
   - EXPAND dimensions to match model input shape
   - PREDICT yhat using model

6. Determine predicted class:
   - IF yhat > 0.5:
       - SET predicted_class = 'before'
     ELSE:
       - SET predicted_class = 'after'
   - PRINT predicted_class

7. Load annotation file:
   - OPEN 'class_train_annotation.json'
   - PARSE JSON data into variable annotations

8. Extract image base name:
   - REMOVE the last part of image_name after the last underscore ('_')
   - STORE the base name in image_base_name

9. Initialize attribute lists:
   - SET before_attributes = []
   - SET after_attributes = []

10. Iterate through annotations:
    - FOR each annotation in annotations:
        - FOR each content in annotation['contents']:
            - IF image_base_name matches content['name'] AND content['name'] ends with predicted_class:
                - FOR each attribute in annotation['attributes']:
                    - REMOVE the question suffix from attribute['question']
                    - IF attribute['answer'] is 'before':
                        - APPEND formatted attribute to before_attributes
                    - ELSE IF attribute['answer'] is 'after':
                        - APPEND formatted attribute to after_attributes
                - BREAK inner loop

11. Print extracted attributes:
    - PRINT "Attributes for Before Image:"
    - FOR each attribute in before_attributes:
        - PRINT attribute
    - PRINT "Attributes for After Image:"
    - FOR each attribute in after_attributes:
        - PRINT attribute
```

(Figure 1.16.1) – Output of the Attribute Classifications + Pseudocode for operation

```
[279]: from tensorflow.keras.metrics import Precision, Recall, BinaryAccuracy
```

```
[281]: pre = Precision()
re = Recall()
acc = BinaryAccuracy()
```

```
[283]: for batch in test.as_numpy_iterator():
    X, y = batch
    yhat = model.predict(X)
    pre.update_state(y, yhat)
    re.update_state(y, yhat)
    acc.update_state(y, yhat)
```

```
WARNING:tensorflow:6 out of the last 11 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x000001EAD3B55620> trigg
red tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tens
rs with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has redu
e_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://ww
w.tensorflow.org/api_docs/python/tf/function for  more details.
WARNING:tensorflow:6 out of the last 11 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x000001EAD3B55620> trigg
red tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tens
rs with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has redu
e_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://ww
w.tensorflow.org/api_docs/python/tf/function for  more details.
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 128ms/step
```

```
[285]: print(f'Precesion:{pre.result().numpy()}, Recall:{re.result().numpy()}, Accuracy:{acc.result().numpy()}')
```

```
Precesion:0.9166666865348816, Recall:0.8148148059844971, Accuracy:0.8793103694915771
```

```
1. Import necessary metrics:
   - Precision
   - Recall
   - BinaryAccuracy

2. Initialize metric objects:
   SET pre = Precision()
   SET re = Recall()
   SET acc = BinaryAccuracy()

3. Iterate over the test dataset:
   FOR each batch in test dataset:
     a. Extract input data (X) and true labels (y)
     b. Use model to predict outputs (yhat) for X
     c. Update Precision metric using y and yhat
     d. Update Recall metric using y and yhat
     e. Update Accuracy metric using y and yhat

4. Retrieve final evaluation results:
   - COMPUTE precision = pre.result().numpy()
   - COMPUTE recall = re.result().numpy()
   - COMPUTE accuracy = acc.result().numpy()

5. Print evaluation results:
   DISPLAY "Precision:", precision
   DISPLAY "Recall:", recall
   DISPLAY "Accuracy:", accuracy
```

(Figure 1.19) – Code to Evaluate Model Performance + Pseudocode