Table of Contents

Table of Contents **2**

1.0 Introduction **3**

1.1 Research Goal and Objectives 4

2.0 Methodology **5**

2.1 Architecture Design 6

2.1.1 Convolutional Layers 6

2.1.2 Pooling Layers 9

2.1.3 Fully Connected & Supporting Layers 11

2.2 Implementation 13

3.0 Training and Evaluation **18**

3.1 Dataset Splitting 18

3.2 Model Compilation 18

3.3 Model Training 19

3.4 Evaluation and Metrics 21

4.0 Conclusion **24**

5.0 References **25**

1.0 Introduction

Nowadays, in the global food supply chain, the assurance of quality and safety of perishable food, especially meat, is vital from both the public health and economic perspectives. Freshness of meat is critical to consumer safety as rotten meat is decayed by bacteria that are capable of causing food-become-borne diseases. Such bacteria not only swept off with the meat's nutritional content but also resulted in serious complications such as stomach infection, dehydration and obstetrics emergencies that may result in death. The consequences of such distribution of rotten or spoiled meat are more serious as it also involves loss of brand image and reputations, great financial losses in terms of recalling of the products and suppliers, and retailers making huge losses.

Visual observation, smell, and chemical tests are commonly used for determining the freshness of meat and as a result, they are not always practical or workable, especially as food moves through more complex supply chains and is transported longer distances before reaching consumers. Therefore, more accurate and fully automated tools for meat spoilage detection are increasingly needed, especially when the aim is to distribute safe and quick recalled products while minimizing food losses. This problem is expected to be overcome intelligently through the integration of computer vision and deep learning techniques related to this project.

The purpose of this project is to demonstrate and enhance the ability of Convolutional Neural Networks (CNNs), a type of deep learning model proven effective for image recognition tasks, which can automatically classify the freshness level of meat based on visual cues. By integrating the "Meat Freshness Image Dataset," which consists of pictures of fresh, half-fresh, and spoiled meat, this project aims to develop a model that can accurately distinguish between these types of meat by examining a variety of visual elements like colour, texture, and surface characteristics. The urgent need to speed up freshness identification is what drove the selection of this dataset. However, by accelerating the speed and accuracy of freshness detection, we can improve food handling procedures, ensuring that consumers receive safer products.

1.1 Research Goal and Objectives

With the intention of enhancing food safety and optimizing quality control processes within the food industry, a smart, AI-driven solution for evaluating meat freshness levels becomes the main objective of this project. Images of meat freshness will be distinguished and categorized by Convolutional Neural Networks (CNNs) in order to achieve the goal. For example, CNNs will analyze the visual characteristics that distinguish fresh, half-fresh, and spoiled meat. Besides, the efficiency and accuracy of CNNs in identifying the meat freshness levels will be evaluated in order to verify that the model can reliably categorize meat under different conditions. Last but not least, this project will investigate the practical application of the model in real-world environments, examining its capability for use in meat processing and retail settings where efficient, accurate freshness detection is crucial.

2.0 Methodology

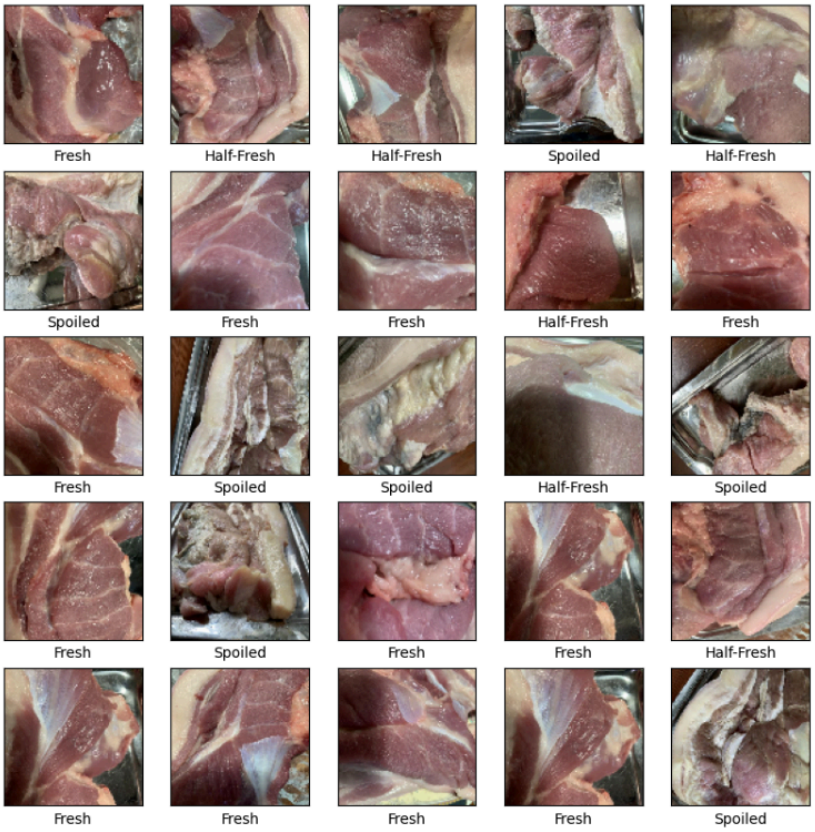

Figure 2.0 Sample data from selected dataset

Our project, *Smart Solutions for Meat Freshness: Ensuring Food Safety with CNN*, addresses concerns in food safety and supply chain management by developing an AI system capable of accurately assessing the freshness of meat. The methods we used to develop our AI system were achieved in the following steps: architecture design and implementation, which we will justify in sections 2.1 and 2.2. The dataset, *Meat freshness image dataset 2022* by Shanawad, V. from Kaggle, consists of 2,226 images of meat categorized into three classes: fresh, half-fresh, and spoiled. Our approach was to use TensorFlow as our framework, as it allows for easy handling of large datasets and model building. Data is split into training and testing sets to evaluate model performance. The CNN architecture was built using the Sequential API in TensorFlow, consisting of convolutional layers, pooling layers, and fully connected layers, splits the images into smaller blocks and extracts features from each block using a specific set of weights. (Anon, *Convolutional Layer* n.d). Figure 2.0 shows a few samples of the dataset in each category.

2.1 Architecture Design

2.1.1 Convolutional Layers

The number of convolutional layers presented in our model is documented here in Table 1. Each layer uses a 3x3 kernel and a ReLU activation function, as for the filters it starts with 32, 64, and finally 128.

Table 2.1 Convolutional Layers

	Filters	Kernel	Activation
First Layer	32	3x3	ReLU
Second Layer	64	3x3	ReLU
Third Layer	128	3x3	ReLU

Justification

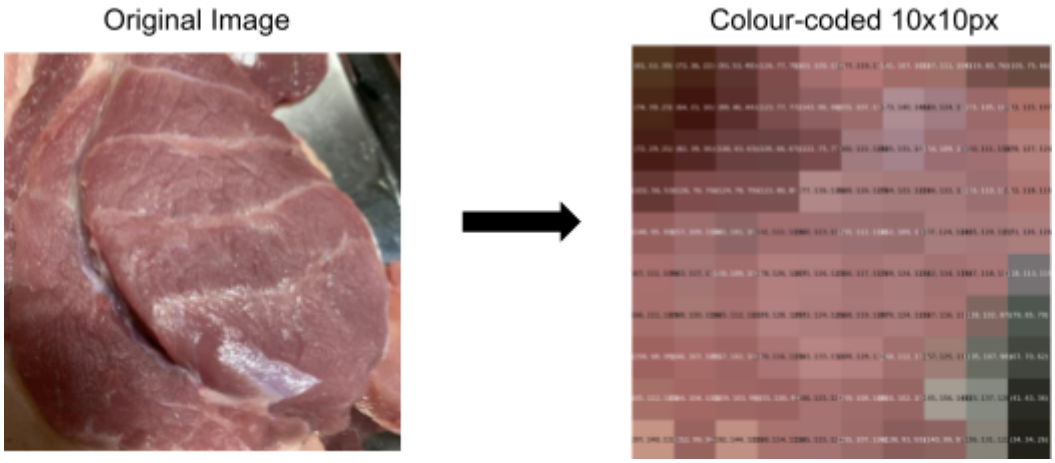

Figure 2.1.1.1 Colour-coding each pixel

To understand how convolutional layers will help in our CNN model, we first need to explore how the convolution operation acts as a feature detector. The process begins with the input, which is the original image, such as the example below. By using Python, we can extract the color code from each pixel of the image, allowing the convolutional layers to apply filters that detect important features such as edges, textures, and patterns. For simplicity of the visualization, we have scaled the pixels down to 10x10px, however in our original model, we are using 128x128px. This is shown in Figure 2.1.1.1.

```
# Define kernels
horizontal_edge_kernel = np.array([[-1, -1, -1],
                                   [ 0,  0,  0],
                                   [ 1,  1,  1]])

vertical_edge_kernel = np.array([[-1, 0, 1],
                                 [-1, 0, 1],
                                 [-1, 0, 1]])

sharpening_kernel = np.array([[ 0, -1,  0],
                              [-1,  5, -1],
                              [ 0, -1,  0]])
```

Figure 2.1.1.2 3x3 Kernel Code Snippet

Now that we have color-coded each pixel, we can begin applying our kernels. For this explanation, we used three types of 3x3 kernels to demonstrate how they work: horizontal edge detection, vertical edge detection, and sharpening. As represented in Figure 2.1.1.2, each kernel focuses on different aspects of the image, with the 3x3 size chosen for its balance between capturing important details and maintaining computational efficiency. By applying these kernels, we enhance key features in the image, which helps the CNN model to learn and detect patterns more effectively during training.

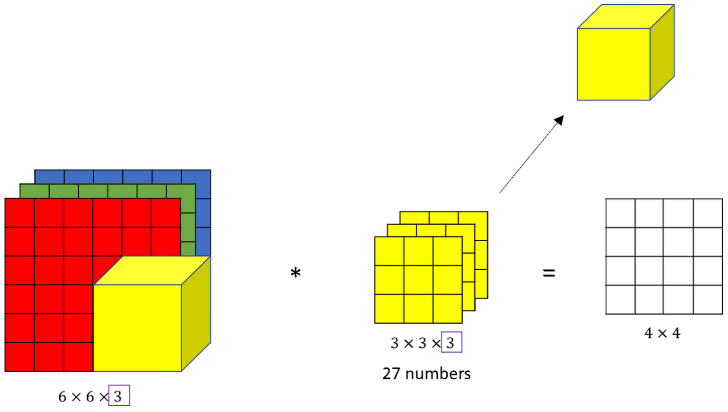

Figure 2.1.1.3 Kernel process of RGB (Datahacker.rs, 2020)

Since our dataset images are not in grayscale, the images will be split into three separate layers corresponding to the red, green, and blue (RGB) channels. Consequently, the kernel must also have three channels, one for each color layer. When the kernel is applied, it will perform convolution across all three channels simultaneously, extracting feature information from the red, green, and blue layers. This means that for each pixel in the image, the kernel will compute separate feature maps for each color channel, then combine them to form the final output.

Using Python, Figure 2.1.1.3 has demonstrated how the kernel has been applied across the RGB channels, the output will be a feature map that highlights certain patterns, edges, or textures identified by the kernel in the image. The output will contain sharper details or highlighted features, depending on the type of kernel used (such as edge detection or sharpening), and will look like a modified version of the original image with the key features emphasized.

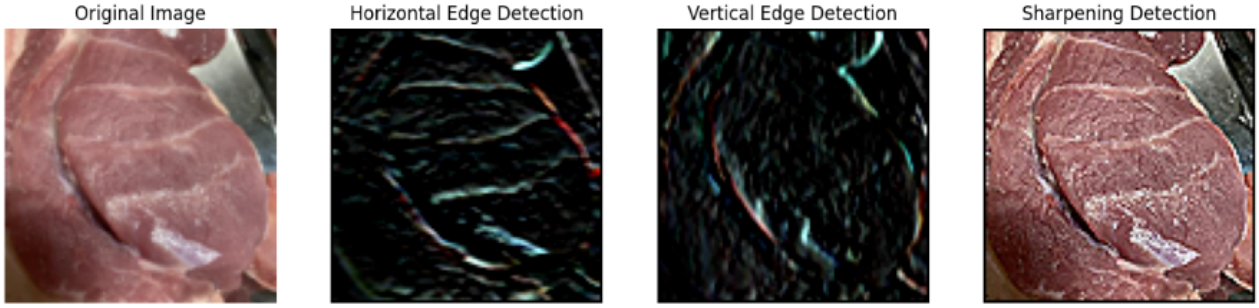

Figure 2.1.1.4 Image after 3x3 Kernel being applied

As we add more filters in our convolutional layers, the model becomes better at recognizing a variety of features in the input data. Each filter focuses on different patterns, such as edges, textures, and shapes. With more filters, the network can learn a richer set of features, which improves its accuracy and overall performance. As the network grows deeper, the additional filters help the model identify more complex and abstract features, which are crucial for understanding intricate patterns or objects. Moreover, having more filters enables the model to generalize better, allowing it to detect subtle differences between classes. By gradually increasing the number of filters to 32, 64, 128, and 256, we significantly enhance the model's ability to identify various patterns and textures. (Lane, 2021)

The choice of an odd-sized kernel, such as 3x3, ensures that the pixels from the previous layer are symmetrically distributed around the output pixel, which helps prevent distortions across layers. In contrast, using an even-sized kernel can introduce aliasing errors, which are distortions in the original input. While aliasing is often mitigated through low-pass filtering in signal processing, this technique cannot be effectively applied in deep networks as it would degrade performance. The preference for smaller kernel sizes like 3x3 is also due to the quadratic growth of parameters with increasing kernel size, which makes larger kernels inefficient. By limiting the number of parameters, the model is encouraged to focus on learning features that

generalize well across different situations. This approach leads to better generalization and improved performance. While smaller kernels, such as 1x1, are excluded due to their inability to capture broader contextual information from neighboring pixels, 3x3 kernels strike a balance between computational efficiency and effective feature extraction. Hence, 3x3 or 5x5 kernels are commonly used in CNNs due to their effectiveness in extracting meaningful patterns without introducing inefficiencies (Ihare, 2020).

2.1.2 Pooling Layers

The number of pooling layers presented in our model is documented into Table 2.2.

Table 2.2 Pooling Layers

Layers	Pooling Type	Pooling Size
First Layer	Max Pooling	2x2
Second Layer	Max Pooling	2x2
Third Layer	Max Pooling	2x2

Justification

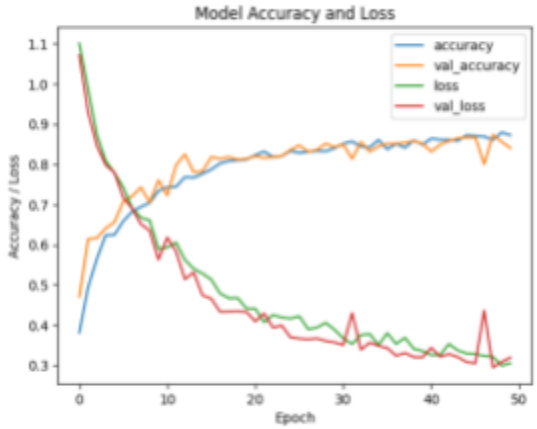

Figure 2.1.2.1(a) Max Pooling in every layer

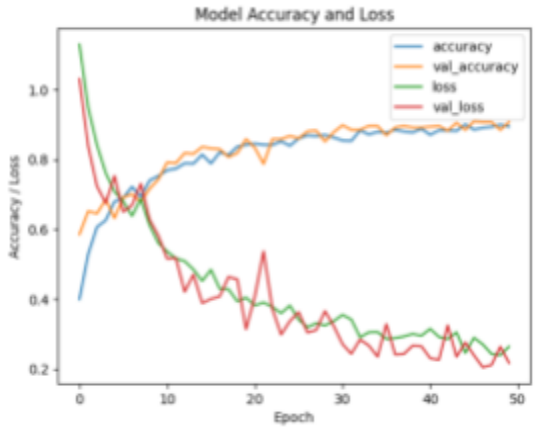

Figure 2.1.2.1(b)Max Pooling only in the first layer

As to why we implemented max pooling layers in each convolutional layer, we initially tested using max pooling in every convolutional layer and found that it resulted in higher accuracy during each epoch compared to using pooling less frequently. Not only it produces a higher accuracy but, comparing the two graphs, Figure 2.1.2.1(a) and Figure 2.1.2.2(b), the chances of overfitting happening if we only add one max pooling layer is higher than if we add max pooling in every layer. We can observe this by looking at the *val_loss* in Figure 2.1.2.1(b),

as it fluctuates more causing it to produce a more uneven result. Therefore, justifying our methods to add max pooling in every convolutional layer.

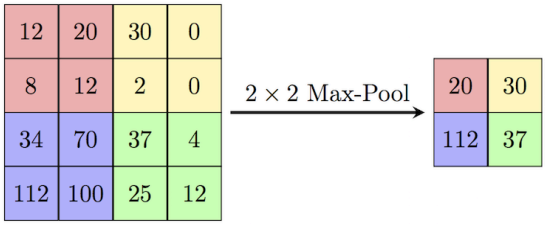

Figure 2.1.2.2 Max Pooling (Anon, 2018)

Pooling layers, particularly max pooling, are essential for reducing the size of the feature maps while preserving the most important features. This reduction not only lowers the computational cost, allowing the model to train faster, but also enhances the key features. In Figure 2.1.2.2, by applying a sliding 2x2 window over the feature map and selecting the maximum value in each non-overlapping region, max pooling sharpens the features by focusing on the most prominent values. This process effectively reduces the size of the feature map while retaining the critical patterns captured by the convolutional layers, making the network more efficient and accurate at recognizing important features. (Lane, 2021)

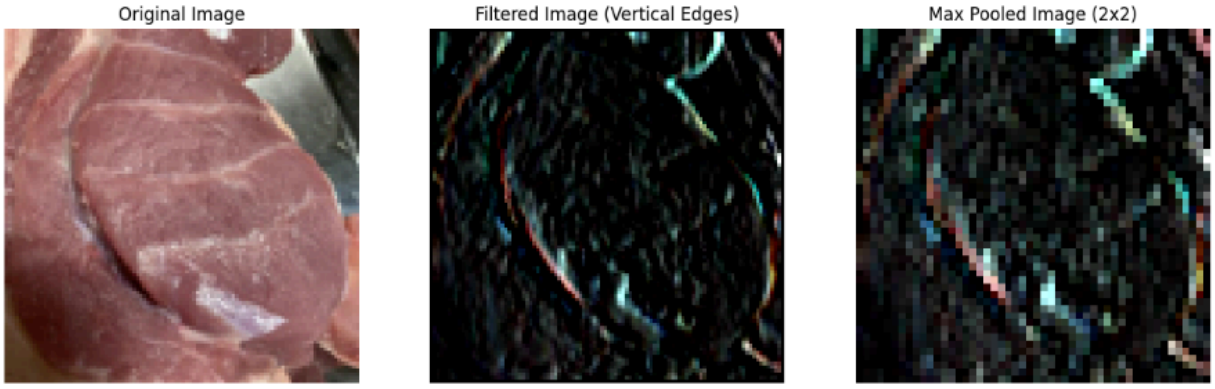

Figure 2.1.2.3 Max Pooling Process

As we add more layers, we reduce the size of the convolutional image while retaining its key features, and simultaneously increase the number of kernel filters. This process aids in precise feature extraction. Figure 2.1.2.3 displays how max pooling operates on a convolutional layer using a vertical edge filter. As shown, max pooling reduces the size of the filtered image while significantly enhancing the features. If we employ X convolutional layers, we will

correspondingly generate *X* max pooling layers, ensuring that important features are preserved and further refined.

2.1.3 Fully Connected & Supporting Layers

The fully connected layers and other supporting layers we used in our model, including dropouts, flatten and dense layers, such details will be displayed in Table 2.1.3.

Table 2.1.3 Fully Connected and Supporting Layers in CNN Model

	Type	Size	Activation
First Convolutional Layer	Dropout	0.2	-
Third Convolutional Layer	Dropout	0.25	-
Flattening Layer	Flatten	-	-
Fully Connected Layer	Dense Dropout Dense	128 0.5 3	relu softmax

Justification

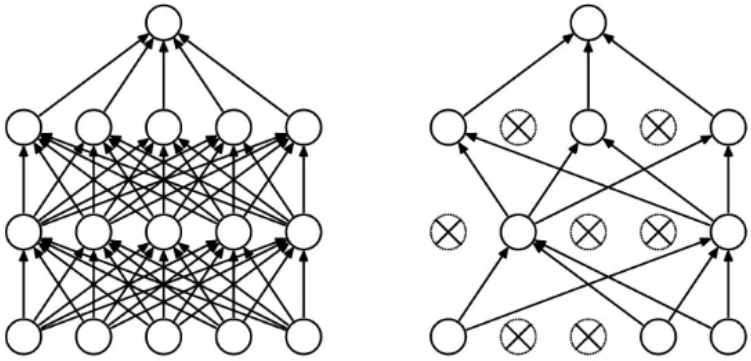

Figure 2.1.3.1 Applying Dropout (Srivastava, 2014)

Dropout layers are used at various points in a neural network to minimize overfitting and improve the model's ability to perform well on new data. This technique works by randomly deactivating a fraction of neurons during each training session, which means the model cannot rely too much on any single neuron or feature. By doing this, dropout helps prevent layers from becoming too dependent on each other to fix errors, leading to a model that is better at handling data it has not seen before. (Yadav, 2022). This step is demonstrated in Figure 2.1.3.1.

Early dropout layers, such as those with a rate of 0.2, are typically applied after the initial layers to manage the complexity of lower-level features. Later in the network, higher dropout rates, such as 0.5, are often used in fully connected layers, where many neurons are active. At a rate of 0.5, half of the neurons in the affected layer are randomly turned off during each training iteration, which further helps in preventing overfitting as these layers are prone to learning overly specific patterns due to their large number of parameters (Hinton et al., 2012).

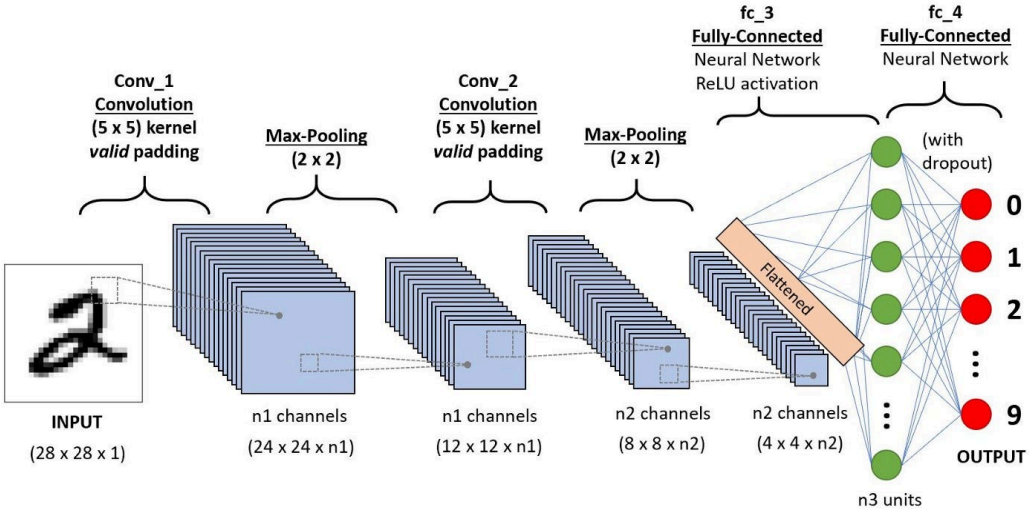

Figure 2.1.3.2 CNN Model Architecture (Ali, 2022)

Figure 2.1.3.2 illustrates the architecture of a Convolutional Neural Network (CNN) model. The Flatten layer serves as a bridge between the convolutional layers, which extract spatial features, and the fully connected layers, which perform classification. By converting 2D feature maps into a 1D vector, the Flatten layer ensures that the spatial features identified in earlier layers are preserved and prepared for use in the dense layers for classification tasks (Goodfellow et al., 2016).

The first dense layer includes 128 units and uses the ReLU activation function, which introduces non-linearity to the model. This allows the network to learn more complex patterns and representations from the extracted features. The final dense layer has three units and applies the softmax activation function, producing probabilities for each class—Fresh, Half-Fresh, and Spoiled. This ensures the output values represent probabilities that sum to one, making the model's predictions easy to interpret.

2.2 Implementation

The CNN model is implemented using TensorFlow, with Keras API for building the layers. Below is a breakdown of the implementation process, including code snippets demonstrating how the CNN model was built, how the data was preprocessed, and how the model was compiled.

1) Libraries

```
import cv2
import os
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns

from tensorflow.keras.models import Sequential
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import load_img, img_to_array, ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical  # for one-hot encoding

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
```

```
from google.colab import drive
drive.mount('/content/drive')
```

Figure 2.2.1. Libraries Code Snippet

This code imports libraries and modules for image processing, data manipulation, model building, and performance evaluation, including TensorFlow for building the CNN, OpenCV for handling images, and Scikit-learn for metrics and preprocessing.

2) Separating Training and Testing images

```
train_images = []
train_labels = []
test_images = []
test_labels = []
```

Figure 2.2.2 Splitting Training and Testing Images Code Snippet

Empty lists such as train_images, train_labels, test_images, and test_labels are initialized to store image data and corresponding labels for training and testing datasets, shown in Figure 2.2.2.

3) Dictionary of Label Encoding

```
class_label_encoding = {
    'SPOILED': 0,
    'HALF-FRESH': 1,
    'FRESH': 2
}
```

Figure 2.2.3 Dictionary of Label Encoding Code Snippet

A dictionary, class_label_encoding is created to map class names (e.g., SPOILED, HALF-FRESH, FRESH) to values (0, 1, 2), making the labels compatible with machine learning models.

4) Data Preprocessing

```
# Function to load images and labels
def load_images_and_labels(dataset_dir, img_size=(128, 128)):
    images = []
    labels = []

    for file_name in os.listdir(dataset_dir):
        if not file_name.lower().endswith(('.png', '.jpg', '.jpeg')):
            continue  # Ignore non-image files

        # Extract class name based on the file name
        if file_name.startswith("SPOILED"):
            class_name = "SPOILED"
        elif file_name.startswith("FRESH"):
            class_name = "FRESH"
        elif file_name.startswith("HALF-FRESH"):
            class_name = "HALF-FRESH"
        else:
            print(f"Skipping file with unexpected class name: {file_name}")
            continue

        # Load and process the image
        img_path = os.path.join(dataset_dir, file_name)
        img = load_img(img_path, target_size=img_size)
        img_array = img_to_array(img) / 255.0   # Normalize the image

        images.append(img_array)

        # Use class_name for labeling
        labels.append(class_label_encoding[class_name])

    return np.array(images), np.array(labels, dtype=int)
```

Figure 2.2.4(a) Data Preprocessing Code Snippet

```
# Define paths for training and testing
train_path = '/content/drive/MyDrive/MeatFreshnessDataset/train/'
test_path = '/content/drive/MyDrive/MeatFreshnessDataset/valid/'

# Load training images and labels
train_images, train_labels = load_images_and_labels(train_path, img_size=(128, 128))

# Load testing images and labels
test_images, test_labels = load_images_and_labels(test_path, img_size=(128, 128))

# Convert labels to one-hot encoding
train_labels = to_categorical(train_labels, num_classes=3)
test_labels = to_categorical(test_labels, num_classes=3)
```

Figure 2.2.4(b) Data Preprocessing 2 Code Snippet

The code in Figure 2.2.4(a) defines a function to load and preprocess images by resizing them to 128x128 px, normalizing pixel values and labeling them based on the file names using a predefined dictionary. This filters non-image files and skips unexpected class names. The function returns the processed images and corresponding labels as NumPy arrays. The training and testing datasets are loaded using this function, with labels converted into one-hot encoded format for use in a multi-class classification task, as reflected in Figure 2.2.4(b).

5) Verifying Data 1

```
# Print shapes of the loaded images
print("Training Images Shape:", train_images.shape)
print("Training Labels Shape:", train_labels.shape)
print("Testing Images Shape:", test_images.shape)
print("Testing Labels Shape:", test_labels.shape)

train_image_count = sum(1 for file_name in os.listdir(train_path) if file_name.lower().endswith(('.png', '.jpg', '.jpeg')))
test_image_count = sum(1 for file_name in os.listdir(test_path) if file_name.lower().endswith(('.png', '.jpg', '.jpeg')))

print(f"Total images in train folder: {train_image_count}")
print(f"Total images in test folder: {test_image_count}")
```

Figure 2.2.5(a) Verifying Data 1 Code Snippet

```
Training Images Shape: (1815, 128, 128, 3)
Training Labels Shape: (1815, 3)
Testing Images Shape: (451, 128, 128, 3)
Testing Labels Shape: (451, 3)
Total images in train folder: 1815
Total images in test folder: 451
```

Figure 2.2.5(b) Verifying Data Output

Initially, there was a problem with the uploading of our dataset where there were missing 400 images in our training folder. Therefore, we added code to print the shapes of the loaded images and labels. This helped us verify that the dataset size and structure were correct before starting the training process.

6) Verifying Data 2

```
class_names = ['Fresh', 'Half-Fresh', 'Spoiled']

# Step 7: Verifying data
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Figure 2.2.6(a) Verifying Data 2 Code Snippet

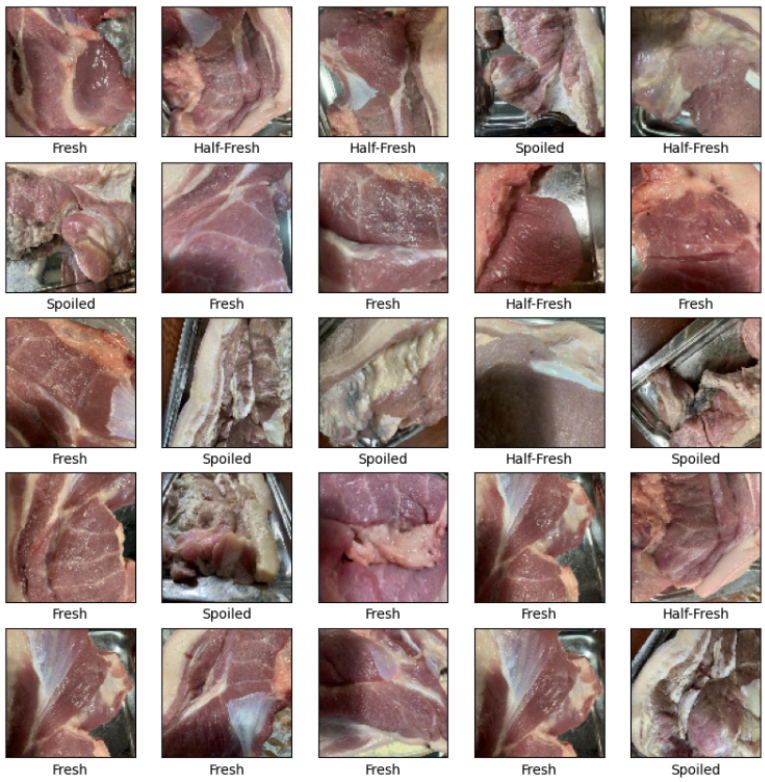

Figure 2.2.6(b) Verifying Data 2 Output

To visually verify the correctness of the dataset, a 5x5 grid of randomly selected training images is displayed using Matplotlib. Each image is shown alongside its corresponding class label, helping to ensure that the data has been loaded and labeled correctly and providing an initial look at the dataset's distribution and content.

7) CNN Model Architecture

```
# Model architecture
model = models.Sequential()

# Convolutional Layers
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.2))

# Second Convolutional Block
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

# Third Convolutional Block
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Dropout(0.25))
model.add(layers.Flatten())

# Fully Connected Layers
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(3, activation='softmax'))  # 3 classes for one-hot encoded labels

model.summary()
```

Figure 2.2.7(a) CNN Model Architecture Code Snippet

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d_3 (MaxPooling2D)	(None, 63, 63, 32)	0
dropout_3 (Dropout)	(None, 63, 63, 32)	0
conv2d_4 (Conv2D)	(None, 61, 61, 64)	18,496
conv2d_5 (Conv2D)	(None, 59, 59, 128)	73,856
dropout_4 (Dropout)	(None, 59, 59, 128)	0
flatten_1 (Flatten)	(None, 445568)	0
dense_2 (Dense)	(None, 128)	57,032,832
dropout_5 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 3)	387

Total params: 57,126,467 (217.92 MB)
Trainable params: 57,126,467 (217.92 MB)
Non-trainable params: 0 (0.00 B)

Figure 2.2.7(b) Model Summary Output

The CNN architecture consists of three convolutional blocks, each with a convolutional layer, followed by max pooling. Dropout layers are added after the first and third blocks to reduce overfitting. After flattening, the network includes fully connected layers with ReLU activation for feature extraction and a final softmax layer for multi-class classification.

3.0 Training and Evaluation

3.1 Dataset Splitting

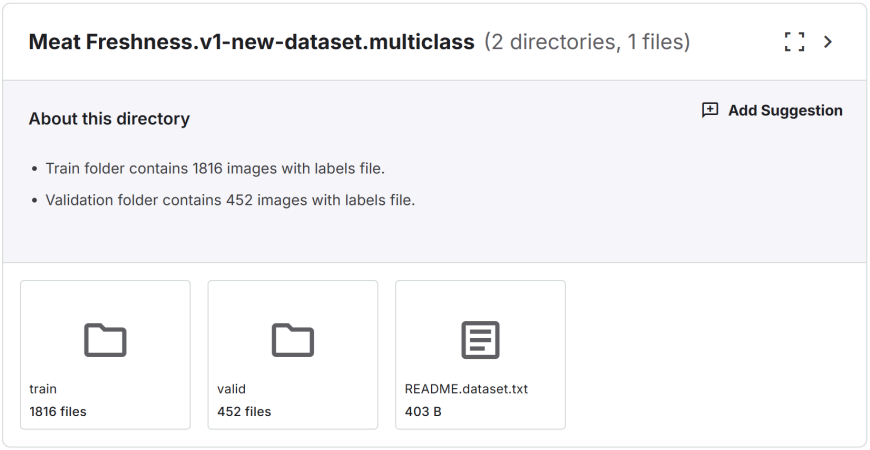

Figure 3.1 Directory of Dataset (Shanawad, 2022)

The dataset was divided into two parts: 20% for testing and 80% for training. The utilization of 80/20 training/testing split ratio is sufficient to provide training samples even for multiclass classification (Rácz, Bajusz and Héberger, 2021). As the splitting is already done in the initial dataset of Shanawad.V, as illustrated in Figure 3.1, hence no further partitioning using tools like *train_test_split()* from the sklearn library is needed.

3.2 Model Compilation

```
# Compile the model
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Figure 3.2.1 Model Compilation Code Snippet

The model was compiled using Adam optimizer with a learning rate of 0.0001, as shown in Figure 3.2. Adam (Adaptive Moment Estimation), a popular choice of optimization, handles sparse gradients efficiently by adapting the learning rate of each parameter (Zaheer and Shaziya, 2019). The *categorical_crossentropy* loss function, especially designed for multi-class classification, was chosen as the target variables in the dataset were preprocessed into one-hot encoded format using *to_categorical()* function. The evaluation metric is set with 'Accuracy', the most straightforward interpretation on the model performance.

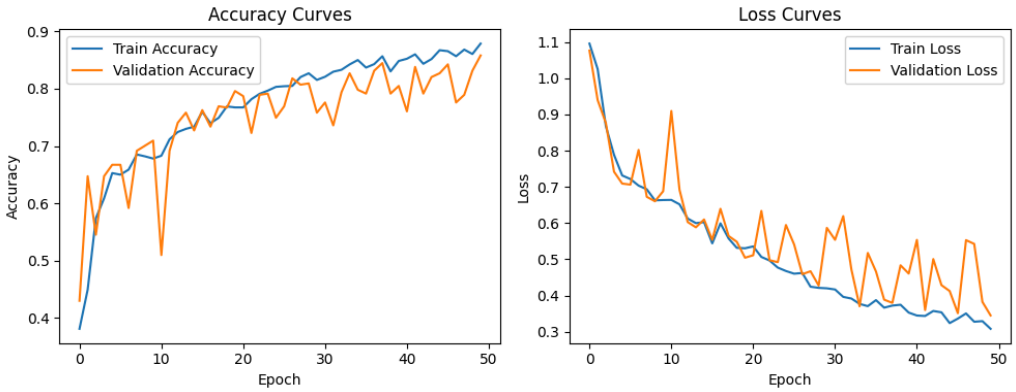

Figure 3.2.2 Output of Learning Curves with 0.001 Learning Rate

Although the Adam optimizer adjusts learning rates automatically, the learning rate of 0.0001 is chosen over the commonly used 0.001, as it showed potential of overfitting during training, as illustrated in Figure 3.2.2. A smaller learning rate helps avoid large updates, which could lead to overshooting the optimal solution. This also stabilizes the training process, reducing the potential for overfitting.

3.3 Model Training

```
# Fit the model
history = model.fit(datagen.flow(train_images, train_labels), batch_size=32,
                    validation_data=(test_images, test_labels),
                    epochs=50,
                    callbacks=[early_stopping])
```

Figure 3.3.1 Training Model Code Snippet

The choice of batch size, epochs, and other hyperparameters for training convolutional neural networks (CNN) for image classification tasks is an important consideration. Referencing Figure 3.3.1, batch size of 32 was chosen to balance between computational efficiency and model performance. By utilizing smaller batch sizes in the range of 4 to 32, it is possible to achieve optimum training stability and generalization performance (Masters and Luschi, 2018). Different values of epochs, such as 50, 10 and 100, were tested, with the results showing that 50 epochs provided the most stable growth in the training process of the model.

Data Augmentation

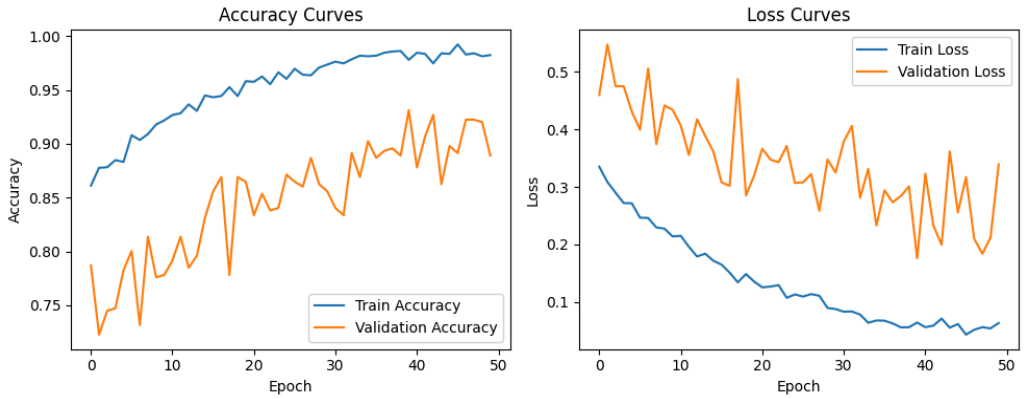

Figure 3.3.2 Learning Curves without Data Augmentation

During the initial training process, the model showed signs of overfitting despite great accuracy, as represented in Figure 3.3.2, where the performance on the training data was better than the validation data. This implies that the model failed to generalize the unseen data, in which additional measures need to be taken to address the issue.

```
# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=25,
    zoom_range=0.2,
    shear_range=0.2,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest'
)
datagen.fit(train_images)
```

Figure 3.3.3 Data Augmentation Code Snippet

Data augmentation techniques were first applied to enhance the diversity of the training dataset. Methods such as random rotation, zoom, flips, and shears were implemented, which effectively improved the model's ability in generalization. These techniques significantly reduced overfitting and enhanced the model's performance.

Early Stopping

```
# Early stopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)
```

Figure 3.3.4 Early Stopping Code Snippet

Early Stopping was implemented as a callback function in this model, as shown in Figure 3.3.4, to prevent overfitting and ensure efficient training. The early stopping mechanism monitored validation loss; if the validation loss failed to improve for 10 consecutive epochs, the training process would be terminated beforehand. This approach prevented excessive training and ensured the model was optimized without overfitting.

3.4 Evaluation and Metrics

The model was evaluated with various metrics, including Accuracy, Precision, Recall, and F1-score, along with visualizations using curve graphs and Confusion Matrix to provide a comprehensive understanding of the performance of the model.

Accuracy

```
# Evaluation on Validation Data
test_loss, test_accuracy = model.evaluate(test_images, test_labels, verbose=0)

print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")
```

```
Test Accuracy: 88.69%
Test Loss: 0.2816
```

Figure 3.4.1 Accuracy Code Snippet and Output

Accuracy is the first metric used to assess overall performance. An accuracy of 88.69% can be obtained from the outcome, indicating a total of 88.69% images can be correctly classified. Meat classification models with good accuracy often fall in the range of 80-90% or higher, which is considered a decent result (Shi *et al.*, 2021). However, more approaches need to be done to get a comprehensive view of the model's performance.

Precision, Recall and F1-Score

```
# Calculate metrics
accuracy = accuracy_score(y_true, y_pred_classes)
precision = precision_score(y_true, y_pred_classes, average='weighted')
recall = recall_score(y_true, y_pred_classes, average='weighted')
f1 = f1_score(y_true, y_pred_classes, average='weighted')

print(f"Accuracy: {accuracy * 100:.2f}%")
print(f"Precision: {precision * 100:.2f}%")
print(f"Recall: {recall * 100:.2f}%")
print(f"F1 Score: {f1 * 100:.2f}%")
```

```
Accuracy: 88.69%
Precision: 89.32%
Recall: 88.69%
F1 Score: 88.80%
```

Figure 3.4.2 Precision, Recall, F1-Score Code Snippet and Output

Precision and Recall provide better insights into how well the model performs on each category. Precision, the proportion of correctly predicted positives out of all predicted positives, was 89.32%, reflecting the model's ability to avoid false predictions. Recall measures the ability to identify actual positives, was 88.69%, showing a decent result in identifying true positives and false negatives. The F1-Score, a harmonic mean of Precision and Recall, was 88.80%, emphasizing the overall effectiveness of the model in balancing the metrics (Grandini, Bagli and Visani, 2020). These values demonstrate the model's performance across categories, and potential improvement when identifying similar samples.

Confusion Matrix

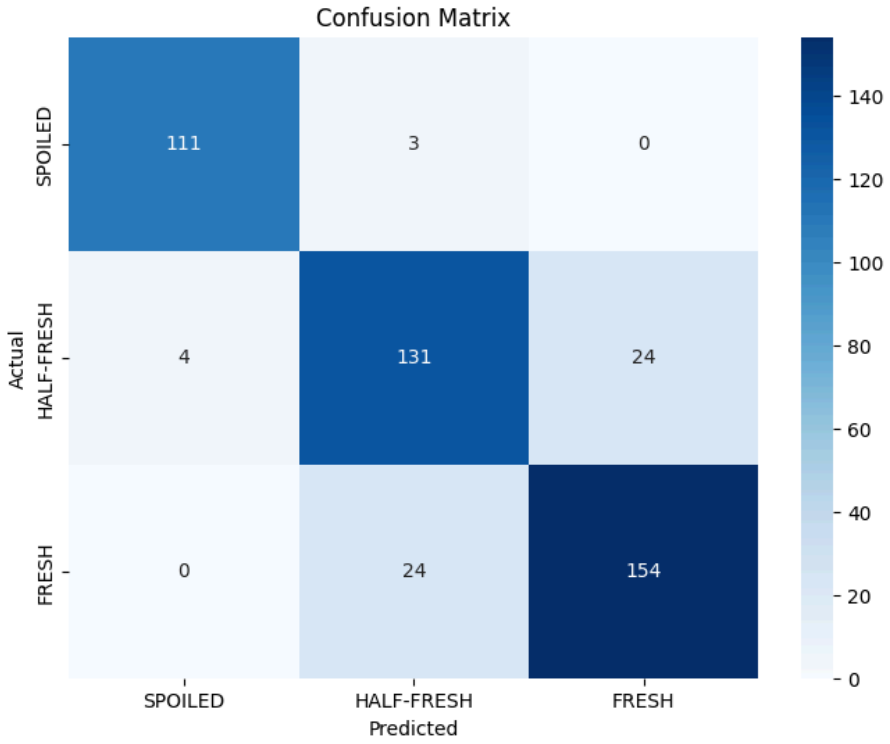

Figure 3.4.3 Confusion Matrix

The confusion matrix, illustrated in Figure 3.4.3, provides a breakdown of the model's performance. This diagonal matrix represents the correctly classified instances for each category: 111 Spoiled samples, 131 Half-Fresh samples, and 154 Fresh samples were correctly identified by the model. Since the majority values are accumulated along the diagonal, this indicates that the model is performing well in differentiating between different meat samples.

However, there are quite a number of misclassifications between the Half-Fresh and Fresh category, with 24 instances from each category being incorrectly classified into the other. This suggests that the model may not be capable of capturing subtle differences between Half-Fresh and Fresh meat, possibly due to the similarity in appearance or texture for both categories.

Learning Curves

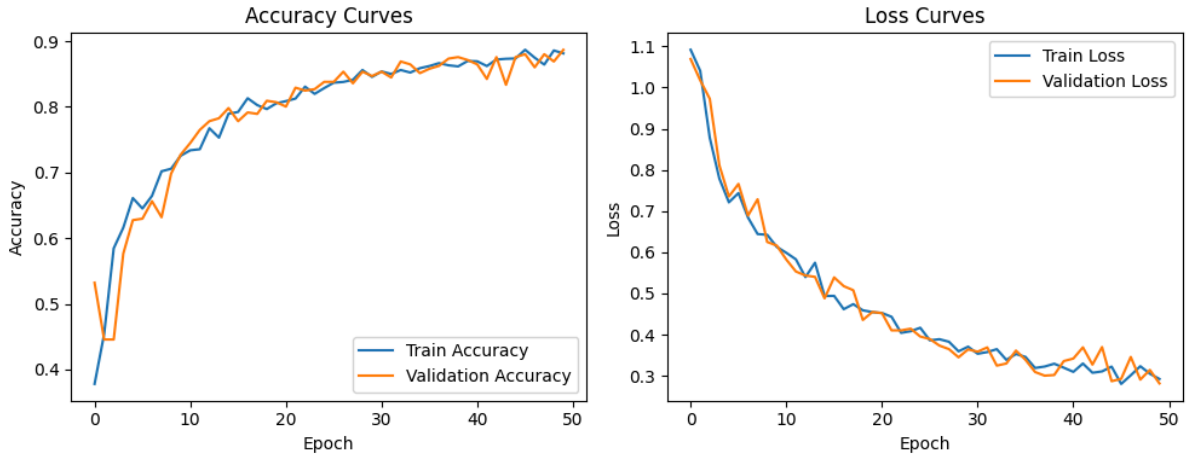

Figure 3.4.4 Learning Curves

Learning curves show the model's training and validation accuracy/loss over 50 epochs. As illustrated in Figure 3.4.4, the training curve indicates a steady increase in accuracy and decrease in loss, indicating effective learning of the model. Around the 20th epoch, both curves start to align closely, suggesting that the model is generalizing well for the unseen data. Overall, the learning curves highlight a well-trained model, with stable performances across training and validation datasets. This concludes that the hyperparameters and training strategies were highly effective on the model.

These results suggest that the implemented architecture and training were effective in achieving a robust classification model. Nevertheless, while the model demonstrates strong performance, there are still possible amendments that could be made for better improvement, particularly in addressing the misclassifications between similar samples to enhance the model's ability in generalizing a more diverse dataset.

4.0 Conclusion

In conclusion, this assignment report presented a completed methodology to developing a Convolutional Neural Network (CNN) for meat freshness classification, which is using the "Meat Freshness Image Dataset" to classify images into three categories: fresh, half-fresh, and spoiled. The main objective was to create an automated, Artificial Intelligence-based system that could improve food safety by efficiently detecting the level of meat freshness, thus improving the quality control processes within the food industry.

The methodology adopted used CNN architecture consisting of convolutional layers, pooling layers, and fully connected layers to extract relevant features from the images. The model was implemented using TensorFlow and Keras, and key strategies such as data augmentation and early stopping were also used to prevent overfitting and improve the model's generalization capabilities. Through this approach, the model is capable of achieving an accuracy of *88.69%*, with high precision, recall, and F1-scores, which indicate strong performance in classifying the meat freshness levels.

However, there are some challenges within the model. The most significant challenge was the misclassification between the fresh and half-fresh meat categories. This was likely due to the subtle or minor visual differences between these two classes, which caused difficulties for the model to capture fully with the available dataset. Despite this, the model demonstrated a high degree of reliability and accuracy, as confirmed by evaluation metrics such as confusion matrices and learning curves. As a result, these visualizations not only helped identify areas for improvement but also showed the importance of further refinement in feature extraction in order to better distinguish between similar categories.

On the other hand, the justification for using CNNs for this group assignment was rooted in their proven effectiveness for image classification. Specifically, using TensorFlow and Keras has proved that they are quite useful in this respect, as they offer flexible and efficient environments in which models can be both built and trained. The decision to implement a multi-layer CNN architecture was also very important in allowing the model to learn both simple and complex features of the meat images, which again will enhance its ability to classify them more accurately.

For future enhancement, we could focus on expanding the dataset to include a wider range of meat images, improving feature extraction techniques, and integrating real-time freshness detection systems for industrial applications. Lastly, with the ability to reduce food waste and enhance consumer health and safety, this project not only demonstrates the potential of AI to improve food safety but also provides a viable solution to those real-world challenges that are faced by the many food industries.

5.0 References

1. *Convolution layer* (no date) *Convolution Layer - an overview | ScienceDirect Topics*. Available at: https://www.sciencedirect.com/topics/computer-science/convolution-layer#:~:text=The%20convolutional%20layer%20splits%20the,strides%2C%20filters%2C%20and%20padding.

2. Datahacker.rs (2020) *#006 CNN convolution on RGB Images*, *Master Data Science*. Available at: https://datahacker.rs/convolution-rgb-image/ (Accessed: 22 October 2024).

3. Dremio. (2024, June 29). *Dropout in neural networks*. https://www.dremio.com/wiki/dropout-in-neural-networks/#:~:text=Dropout%20in%20Neural%20Networks%20operates,to%20generalize%20to%20new%20data.

4. *File:MaxpoolSample2.png* (2018) *Computer Science Wiki*. Available at: https://computersciencewiki.org/index.php/File:MaxpoolSample2.png (Accessed: 22 October 2024).

5. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.

6. Grandini, M., Bagli, E. and Visani, G. (2020) *Metrics for Multi-Class Classification: an Overview arXiv.org*. August 13, 2020 [online]. Available from: https://arxiv.org/abs/2008.05756 [Accessed 27 November 2024].

7. Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. Available at: https://doi.org/10.48550/arXiv.1207.0580

8. Ihare, A. (2020) *Significance of kernel size*, *Medium*. Available at: https://medium.com/analytics-vidhya/significance-of-kernel-size-200d769aecb1#:~:text=Limiting%20the%20number%20of%20parameters,size%20at%203x3%20or%205x5.

9. Lane, C. (2021) *Convolution Operation in CNN*, *YouTube*. Available at: https://youtu.be/E5Z7FQp7AQQ?si=NJewCJxtqt7H3_Ua (Accessed: 22 October 2024).

10. Lane, C. (2021b) *Max Pooling in Convolutional Neural Network*, *YouTube*. Available at: https://www.youtube.com/watch?v=zg_AA3fZpE0&list=PLuhqtP7jdD8CD6rOWy20INGM44kULvrHu&index=5&ab_channel=CodingLane (Accessed: 22 October 2024).

11. Masters, D. and Luschi, C. (2018) *Revisiting small batch training for deep neural networks arXiv.org*. April 20, 2018 [online]. Available from: https://arxiv.org/abs/1804.07612 [Accessed 26 November 2024].

12. Rácz, A., Bajusz, D. and Héberger, K. (2021) Effect of Dataset Size and Train/Test Split Ratios in QSAR/QSPR Multiclass Classification. *Molecules* [online]. 26 (4). [Accessed 26 November 2024].

13. Seldon (2023) *What is covariate shift?*, *Seldon*. Available at: https://www.seldon.io/what-is-covariate-shift#:~:text=Covariate%20shift%20is%20a%20specific,or%20labels%20remain%20the%20same. (Accessed: 22 October 2024).

14. Shanawad, V. (2022) *Meat freshness image dataset*, *Kaggle*. Available at: https://www.kaggle.com/datasets/vinayakshanawad/meat-freshness-image-dataset/data. (Accessed: 22 October 2024).

15. Shi, Y., Wang, X., Borhan, M.S., Young, J., Newman, D., Berg, E. and Sun, X. (2021) A Review on Meat Quality Evaluation Methods Based on Non-Destructive Computer Vision and Artificial Intelligence Technologies. *Food science of animal resources* [online]. 41 (4), pp. 563–588. [Accessed 27 November 2024].

16. Zaheer, R. and Shaziya, H. (2019) *A Study of the Optimization Algorithms in Deep Learning IEEE Xplore*. January 11, 2019 [online]. Available from: https://ieeexplore.ieee.org/abstract/document/9036442 [Accessed 27 November 2024].