


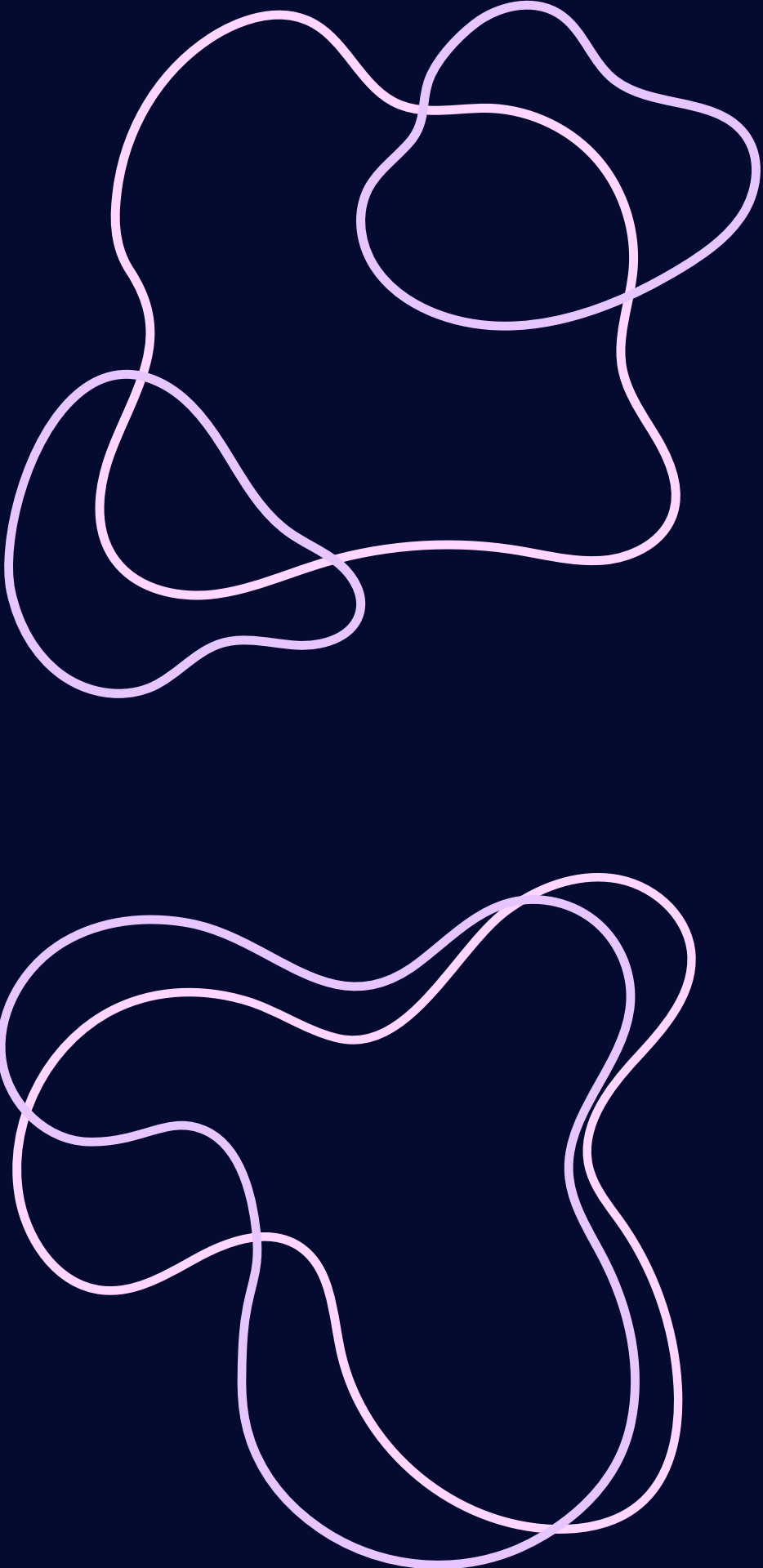
PROJECT EXHIBITION-2 (REVIEW 2)

Detection of defective potato chips


A group project by Team 11

Abstract

Fresh potatoes are being replaced by processed, added-value foods in the global potato food supply. Many different products are made from potatoes after they have been processed, such as cooked potatoes, par-fried potato strips, french fries, chips, potato starch, potato granules, potato flakes, and dehydrated diced potatoes. Very thinly sliced raw potatoes are fried to a final oil and moisture content of around 35% and 1.8%, respectively, to create potato chips. Each batch of potato tubers used to make potato chips must be examined for quality before processing, and the appearance is obviously very important. Consumers base their first assessment of a product's quality on its colour, which is crucial to the item's adoption even before it is tasted. On the other hand, acrylamide, which is produced when potatoes are fried and is closely associated to the colour of the potato chips, has been identified as a significant compound for human health (carcinogenic in rats).



Historically, the colour of potato chips has been evaluated objectively using specialised tools called colorimeters. Some computer vision systems have been testing the online evaluation of potato chips in some European facilities, enabling chips to be categorised according to flaws like blisters or black spots. Additionally, some researchers have been working on an innovative system that has the potential to both categorise chips based on colour and forecast acrylamide levels using neural networks. The object of this presentation is to apply Artificial Intelligence to make a model using a Convolutional Neural Network that classifies a potato chip into defective and non-defective based on image provided to it.





Problem Statement

Potato chip manufacturers spend a lot of time, money and manpower into detection and exclusion of defective potato chips and this whole process could be made a lot more efficient if it was automated with the help of AI.

Objective

Applying Artificial Intelligence to make a model using a Convolutional Neural Network that classifies a potato chip into defective and non-defective based on image provided to it.



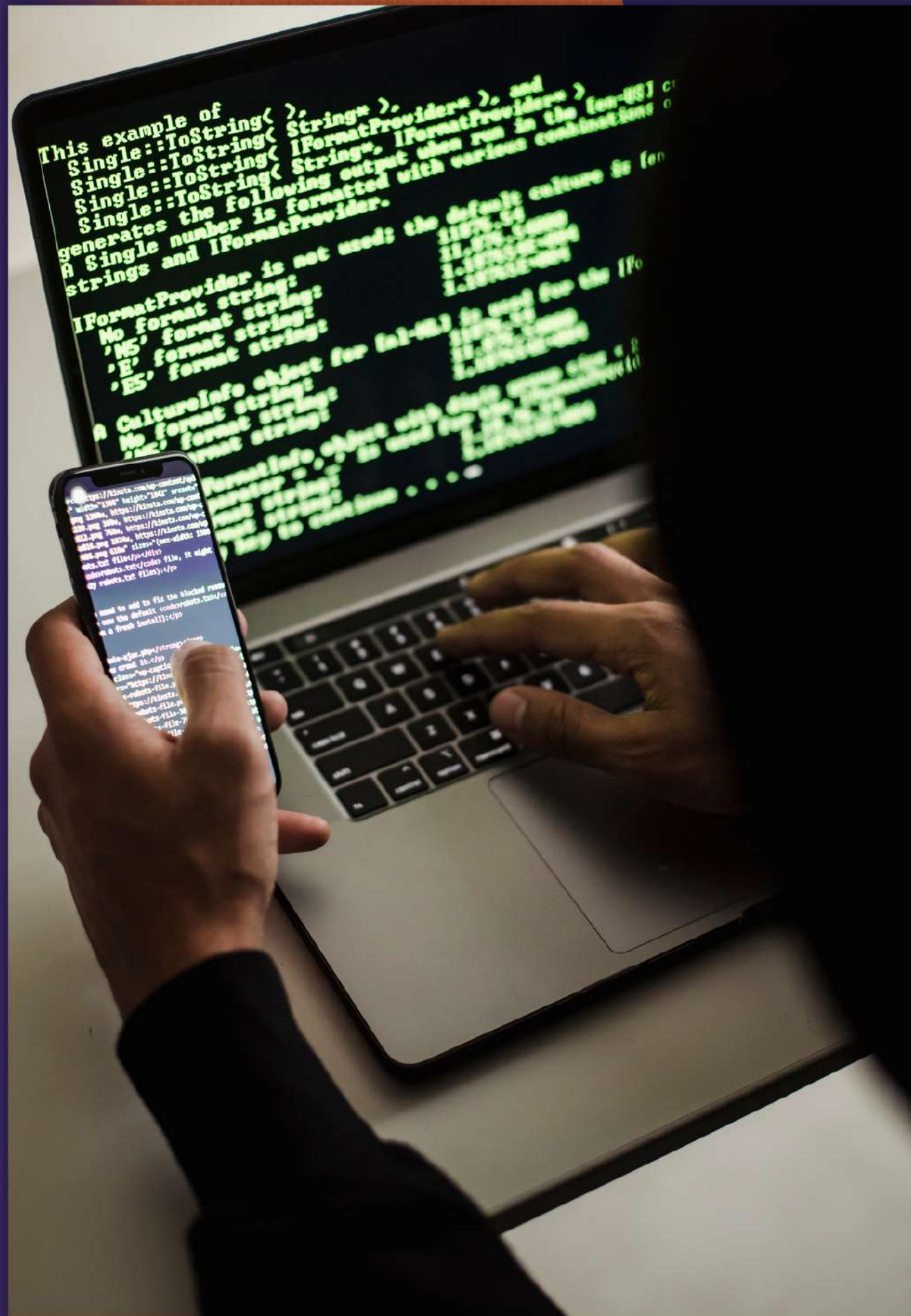
Hardware Requirement

While projects using ML require a powerful GPU, we are going to be leveraging the power of cloud servers (Google Colab) to make the task easier for us, hence any device is enough to run the code.

In case we are developing a GUI for the detection software, we would import the code in a local environment and hence that desktop would need to have a dedicated GPU to run the code smoothly.

Software Requirement

We are going to be using Google Collab to write and run our code since it provides access to free Graphical Processing Units (GPUs) and this would greatly help improve the efficiency of our project.




Flow chart

Gather and annotate a collection of photographs of potato chips. We categorise the pictures as either faulty or nondefective in order to designate the data as having both effective and nondefective chips.

Image processing before procedure can entail scaling the photos to a standard size, making them grayscale, and using any necessary filters or adjustments to improve the chip's characteristics.

We create training and test sets from the data collection. The test set is used to assess how well the CNN performed after being trained using the training set.

We create the CNN architecture, choosing the number of levels, the kind of layers to employ, and the size of the input and output layers.



In order to increase performance, train the CNN using the training set, which involves filling in the input and just the network. The CNN is then given new weights by applying an optimization technique and loss function.

By submitting test set pictures to the network and contrasting the predicted labels with the actual labels, one may assess CNN on the test set.

The CNN's operation If the network's performance is unsatisfactory, the network design, the training data, or the training parameters may need to be adjusted in order to enhance the network's performance.

Code

```
import os

base_dir = 'chips'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')

# Directory with training defective/nondefective pictures
train_defective_dir = os.path.join(train_dir, 'Defective')
train_nondefective_dir = os.path.join(train_dir, 'Non-Defective')

# Directory with validation defective/nondefective pictures
validation_defective_dir = os.path.join(validation_dir, 'Defective')
validation_nondefective_dir = os.path.join(validation_dir, 'Non-Defective')

train_defective_fnames = os.listdir( train_defective_dir )
train_nondefective_fnames = os.listdir( train_nondefective_dir )
```

✓ 0.6s Python

This code is making different folders to put pictures of defective and non-defective chips in. The "train" folder is for pictures that will be used to help the computer learn what a defective and non-defective chip looks like. The "validation" folder is for pictures that will be used to test if the computer learned correctly. The pictures in the "train" folder will have the label "Defective" or "Non-Defective" to tell the computer which is which. The pictures in the "validation" folder will also be labeled "Defective" or "Non-Defective" so the computer can check if it learned correctly. The computer will use the pictures to learn and then check if it learned correctly using the validation set.

Code

```
%matplotlib inline

import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Parameters for our graph; we'll output images in a 4x4 configuration
nrows = 4
ncols = 4

pic_index = 0 # Index for iterating over images
```

✓ 0.6s

Pythor

This code is setting up a way to display pictures using a library called matplotlib. It creates a grid with 4 rows and 4 columns, and it creates a variable called `pic_index` that starts at 0. This variable will be used to keep track of which picture we're looking at when we go through a group of pictures.

Code

```
fig = plt.gcf()
fig.set_size_inches(ncols*4, nrows*4)

pic_index+=8

next_defective_pix = [os.path.join(train_defective_dir, fname)
                      for fname in train_defective_fnames[ pic_index-8:pic_index]
                      ]

next_nondefective_pix = [os.path.join(train_nondefective_dir, fname)
                         for fname in train_nondefective_fnames[ pic_index-8:pic_index]
                         ]

for i, img_path in enumerate(next_defective_pix+next_nondefective_pix):
    # Set up subplot; subplot indices start at 1
    sp = plt.subplot(nrows, ncols, i + 1)
    sp.axis('Off') # Don't show axes (or gridlines)

    img = mpimg.imread(img_path)
    plt.imshow(img)

plt.show()

✓ 10.8s
```

Pyth

This code is creating a figure (a space for an image) that is 4 columns and 4 rows. It then sets the size of the figure in inches. The code is also selecting 8 pictures, 4 from the defective folder and 4 from the non-defective folder, and storing their file paths in two separate lists. Then, it is using a loop to go through the two lists of file paths and displaying the images one by one on the figure, starting with the first image in the first subplot and ending with the last image in the last subplot. The axis of the subplot is also set to "off" to not show any gridlines or coordinates. The final line of the code will display the figure with the images.

Code

```
import tensorflow as tf

model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image 150x150 with 3 bytes color
    tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(150, 150, 3)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # Flatten the results to feed into a DNN
    tf.keras.layers.Flatten(),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    # Only 1 output neuron. It will contain a value from 0-1 where 0 for 1 class ('non-defective') and 1 for the other ('defective')
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

✓ 2.9s

Python

This code defines a neural network model using the TensorFlow library. The model is set up as a sequential model, meaning that the layers are added one by one in the order they are defined. The model consists of several layers including convolutional layers, max pooling layers, a flatten layer and dense layers (fully connected layers). The convolutional layers are responsible for detecting patterns in images, the max pooling layers are used to reduce the spatial dimensions of the image and make the model more robust to small translations of the image. The dense layers are used to learn non-linear combinations of the features detected by the convolutional layers. The final output of the model is a sigmoid activation function, which gives a probability score of the chip being defective.

Code

```
from tensorflow.keras.optimizers import RMSprop

model.compile(optimizer=RMSprop(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics = ['accuracy'])
```

✓ 0.4s

Python

This code is used to configure the model for training. The "compile" function is used to set the optimizer, loss function and metrics for the model. The optimizer being used here is RMSprop with a learning rate of 0.001. The loss function being used is binary_crossentropy and the metric being used to evaluate the model is accuracy. The optimizer and loss function are used to optimize and evaluate the model while training. The metric is used to evaluate the model's performance on the validation set.

Code

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255.
train_datagen = ImageDataGenerator( rescale = 1.0/255. )
test_datagen  = ImageDataGenerator( rescale = 1.0/255. )

# -----
# Flow training images in batches of 20 using train_datagen generator
# -----
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=20,
                                                    class_mode='binary',
                                                    target_size=(150, 150))

# -----
# Flow validation images in batches of 20 using test_datagen generator
# -----
validation_generator = test_datagen.flow_from_directory(validation_dir,
                                                         batch_size=20,
                                                         class_mode = 'binary',
                                                         target_size = (150, 150))
```

✓ 0.6s

Python

This code is setting up the data generators for our model. The ImageDataGenerator class is used to rescale the images by 1./255. It also sets up the batch sizes and target sizes for the training and validation data. The flow_from_directory method is used to read the images from the folders and generate batches of image data. The class_mode is set to 'binary', since we only have 2 classes (defective and non-defective). The train_generator will be used for training our model and validation_generator will be used for testing the performance of the model.

Code

```
history = model.fit(  
    train_generator,  
    epochs=15,  
    validation_data=validation_generator,  
    verbose=2  
)
```

✓ 9m 24.4s

Python

This code is using the "model.fit" function from Tensorflow's Keras library to train the model with the data from the "train_generator" using 15 epochs. An epoch is a measure of the number of times all of the training vectors are used once to update the weights. The "validation_data" is set to the "validation_generator" which will be used to evaluate the performance of the model during training. The "verbose" argument is set to 2 which will make the output of the training process more detailed. The "history" object returned by this function will contain information about the training process, such as the training and validation loss and accuracy, which can be used for visualizing the performance of the model.

Code Output

```
Epoch 1/15
39/39 - 42s - loss: 0.7854 - accuracy: 0.5345 - val_loss: 0.6910 - val_accuracy: 0.4792 - 42s/epoch - 1s/step
Epoch 2/15
39/39 - 39s - loss: 0.6400 - accuracy: 0.6840 - val_loss: 0.3477 - val_accuracy: 0.9167 - 39s/epoch - 990ms/step
Epoch 3/15
39/39 - 38s - loss: 0.2674 - accuracy: 0.8947 - val_loss: 0.1412 - val_accuracy: 0.9479 - 38s/epoch - 962ms/step
Epoch 4/15
39/39 - 36s - loss: 0.1101 - accuracy: 0.9597 - val_loss: 0.0821 - val_accuracy: 0.9635 - 36s/epoch - 932ms/step
Epoch 5/15
39/39 - 36s - loss: 0.1152 - accuracy: 0.9636 - val_loss: 0.0966 - val_accuracy: 0.9740 - 36s/epoch - 923ms/step
Epoch 6/15
39/39 - 36s - loss: 0.0616 - accuracy: 0.9844 - val_loss: 0.1234 - val_accuracy: 0.9635 - 36s/epoch - 923ms/step
Epoch 7/15
39/39 - 42s - loss: 0.0531 - accuracy: 0.9844 - val_loss: 0.0580 - val_accuracy: 0.9948 - 42s/epoch - 1s/step
Epoch 8/15
39/39 - 36s - loss: 0.1180 - accuracy: 0.9701 - val_loss: 0.0243 - val_accuracy: 0.9948 - 36s/epoch - 926ms/step
Epoch 9/15
39/39 - 36s - loss: 0.0518 - accuracy: 0.9857 - val_loss: 0.0599 - val_accuracy: 0.9792 - 36s/epoch - 922ms/step
Epoch 10/15
39/39 - 40s - loss: 0.0868 - accuracy: 0.9805 - val_loss: 0.1340 - val_accuracy: 0.9531 - 40s/epoch - 1s/step
Epoch 11/15
39/39 - 36s - loss: 0.0356 - accuracy: 0.9883 - val_loss: 0.1124 - val_accuracy: 0.9688 - 36s/epoch - 928ms/step
Epoch 12/15
39/39 - 36s - loss: 0.0405 - accuracy: 0.9870 - val_loss: 0.0310 - val_accuracy: 0.9948 - 36s/epoch - 912ms/step
Epoch 13/15
...
Epoch 14/15
39/39 - 38s - loss: 0.0465 - accuracy: 0.9844 - val_loss: 0.0732 - val_accuracy: 0.9792 - 38s/epoch - 980ms/step
Epoch 15/15
39/39 - 37s - loss: 0.0336 - accuracy: 0.9922 - val_loss: 0.0667 - val_accuracy: 0.9844 - 37s/epoch - 945ms/step
```

This output shows the progress of the training process of the model. Each line represents one iteration, also called an "epoch", through the entire training dataset. The model's performance is evaluated using the validation dataset after each epoch.

The columns in the output are as follows:

- **Epoch:** The current number of the epoch.
- **loss:** The loss value for the training dataset at the current epoch. The loss is a measure of how well the model is doing on the training dataset. Lower values indicate better performance.
- **accuracy:** The accuracy value for the training dataset at the current epoch. It is the ratio of the number of correctly predicted classes to the total number of classes. Higher values indicate better performance.
- **val_loss:** The loss value for the validation dataset at the current epoch.
- **val_accuracy:** The accuracy value for the validation dataset at the current epoch.

As the output shows, the accuracy of the model improves as the number of epochs increases, and it is reaching around 99% accuracy. The validation loss and accuracy are also getting better as the number of epochs increases, which means that the model is not overfitting.

Timeline

Project Review - 2

Tentative Target:

Training and creating the CNN model that can classify the defective and non-defective potato chips.

1

Time span

26th-31st Dec 2022

Project Review - 1

Tentative Target:

Basic Framework of The Idea and the steps to produce the model

2

Time span

23rd-27th Jan 2023

3

Time span

13th-16th Feb 2023

Project Review - 3

Tentative Target:

Completing the whole code and making it more efficient wherever possible.

Application

The Potato Chip Industry is worth about 30 Billion Dollars and has been seeing a stable growth in the last decade, all of the major companies spend millions of dollars into Quality Control and this factor ultimately decides the quality of their product and hence their net profit.

An AI Model that could automate this task to make it efficient and cheap is in great demand and would be an invaluable asset to these companies.

ILFORD

3372-11

D HP5 PLUS

3 8A 11



Contribution

All 5 members of this group are going to be contributing towards the code in every step of the process, since that would not only improve our teamwork skills, but it would also give all of us a deeper knowledge of the whole code.



References

<https://www.kaggle.com/datasets/concaption/pepsico-lab-potato-quality-control>

<https://www.google.com/>



THANK YOU
SO MUCH