

PATTERN SENSE:Classification of poultry Diseases for Enhanced

❖ INTRODUCTION:

- Project Title: transfer learning-based classification of poultry diseases for enhanced health management
- Team Members:
 - 1.Vinukonda karimulla(Gathering the complete project related Information)
 - 2.Gorantla Dileep chowdary(Installation of the softwares related to front- End and Back-End)
 - 3.Komera Himani(Importing the libraries and Building the Model)
 - 4.Modepalli Jayanth(Running & Evaluating the Mode)



PROJECT OVERVIEW:

Skills Required:

Python,Data Preprocessing Techniques,Tensorflow,HTML,CSS,JAVASCRIPT

Project Description:

Scenario 1 - Description:

Poultry farming is a vital sector in agriculture, but disease outbreaks can cause significant economic loss and threaten food supply chains. Traditionally, disease identification in birds relies on manual inspection, which is time-consuming and requires expert knowledge. Delays in diagnosis can lead to rapid spread and high mortality among poultry.

This project aims to solve this real-world problem by using Transfer Learning, a powerful technique in Deep Learning, to develop an intelligent system that can automatically detect and classify poultry diseases from images of infected birds. By leveraging pre-trained convolutional neural networks (CNNs), the system can achieve high accuracy even with limited data, enabling early detection and better disease management.



Scenario 2: Technical and Development-Focused Description

Scenario 2 - Description:

This project focuses on implementing a deep learning-based poultry disease classification system using Transfer Learning. The system is built by fine-tuning a pre-trained CNN model (such as VGG16, ResNet50, or MobileNet) on a custom dataset of poultry images showing various diseases like Newcastle Disease, Fowl Pox, and Infectious Bronchitis.

The model is developed using Python, TensorFlow/Keras, and OpenCV for preprocessing and classification tasks. The dataset is augmented to improve model generalization, and performance is evaluated using accuracy, precision, recall, and confusion matrix metrics.

A Flask-based web interface is also provided where users can upload an image of a bird, and the system will predict the disease category. This smart solution can be used in farms or veterinary clinics to assist in early disease detection and improve overall poultry health management.

To accomplish this, we have to complete all the activities and tasks listed below

- Data Collection.
- Create a data frame with image paths, images, and labels
- Split into train_df, test_df, and valid_df
- Data Pre-processing.
- Import the required library
- Apply data augmentation to balance train_df
- Configure ImageDataGenerator class
- Apply ImageDataGenerator functionality to train_df, valid_df, and test_df
- Model Building
- Pre-trained CNN model as a Feature Extractor
- Adding Dense Layer
- Configure the Learning Process
- Train the model
- Save the Model
- Test the model
- Application Building
- Create an HTML file
- Build Python Code



PROJECT STRUCTURE:

- The Data folder contains the training and testing images for training our model.
- We are building a Flask Application that needs HTML pages stored in the templates.
- folder and a python script app.py for server-side scripting
- we need the model that is saved and the saved model in this content is model_cnn (2).h5
- templates folder contains home.html, predict.html & predictionpage.html pages.

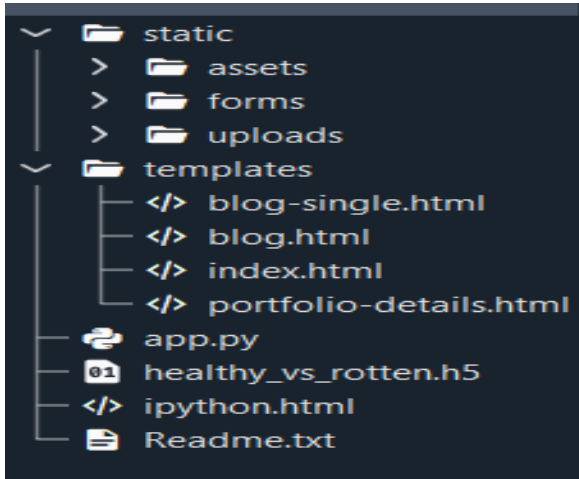
❖ ARCHITECTURE & SETUP INSTRUCTIONS:

- Data collection:

It is the most crucial aspect that makes algorithm training possible. There are many popular open sources for collecting the data.

Eg: kaggle.com, UCI repository, etc.

Create the Project folder which contains files as shown below



- We are building a Flask application with HTML pages stored in the templates folder and a Python script app.py for scripting.
- Healthy_vs_rotten.h5 is our saved model. Further, we will use this model for flask integration.

In this project, we have used 'Common Fabric Pattern' data. This data is downloaded from kaggle.com. It is the most crucial aspect that makes algorithm training possible. So this section allows you to download the required dataset.

There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc. In this project, we have used 'Common Fabric Pattern' data. This data is downloaded from kaggle.com. Please refer to the link given below to download the dataset.

Link: <https://www.kaggle.com/datasets/nguyngiabob/dress-pattern-dataset>

As the dataset is downloaded. Let us read and understand the data properly with the help of some visualization techniques and some analyzing techniques.

Note: There are several techniques for understanding the data. But here we have used some of it. In an additional way, you can use multiple techniques.

We are going to build our training model on Google Colab.

Upload the dataset into Google Drive and connect the Google Colab with drive using the below code

```
▼ Load Image set into colab

✓ 24s from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

✓ 9s [3] !unzip /content/drive/MyDrive/patterns.zip

inflating: raw_data/polka-dotted/polka-dotted_0000105.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000106.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000107.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000108.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000109.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000110.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000111.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000112.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000113.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000114.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000115.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000116.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000117.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000118.jpg
inflating: raw_data/polka-dotted/polka-dotted_0000119.jpg
inflating: raw_data/striped/striped_0000000.jpg
inflating: raw_data/striped/striped_0000001.jpg
inflating: raw_data/striped/striped_0000002.jpg
```

To build a DL model we have six classes in our dataset. But In the project dataset folder training and testing data are needed. So, in this case, we just have to assign a variable and pass the folder path to it.

Importing the libraries

```
[1] import pandas as pd
import numpy as np
from numpy import random
import os
import matplotlib.pyplot as plt
```

Split into train, test, and valid sets

```
[ ] from sklearn.model_selection import train_test_split

[ ] train_df, dummy_df=train_test_split(df, train_size=.8, random_state=123, shuffle=True, stratify=df['label'])
    valid_df, test_df=train_test_split(dummy_df, train_size=.5, random_state=123, shuffle=True, stratify=dummy_df['label'])
    print("Train dataset : ",len(train_df),"Test dataset : ",len(test_df),"Validation dataset : ",len(valid_df))
    train_balance=train_df['label'].value_counts()
    print('Train dataset value count: \n',train_df['label'].value_counts())

Train dataset : 672 Test dataset : 84 Validation dataset : 84
Train dataset value count:
 animal-print    96
 striped        96
 plain          96
 paisley        96
 zigzagged      96
 chequered      96
 polka-dotted   96
Name: label, dtype: int64
```

The code performs data splitting using the `train_test_split` function from scikit-learn. Initially, it splits the original dataset `df` into training (`train_df`) and test (`dummy_df`) sets, allocating 80% of the data to training and 20% to testing, while ensuring class balance through stratification based on the 'label' column. Then, it further divides the test set `dummy_df` into validation (`valid_df`) and final test (`test_df`) sets, each comprising 50% of the data. Finally, it prints the sizes of the three datasets and the value counts of the classes in the training set to verify the distribution.

Extracting labels from the files directory

```
✓ [4] directory = '/content/raw_data'
```

▼ Saving Labels

```
✓ [5] labels = os.listdir(directory)
    labels
```

```
['polka-dotted',
 'plain',
 'chequered',
 'paisley',
 'animal-print',
 'striped',
 'zigzagged']
```

```
✓ [6] labels.sort()
```

This code snippet retrieves a list of labels or classes from a directory. It uses the `os.listdir()` function to get the list of all items (files and directories) in the specified `directory`. The `labels` variable then stores this list, which typically represents the class names or categories used in a classification task. Each item in the `labels` list corresponds to a unique class or category in the dataset. We then sort the list in alphabetical order to ensure that our model gives the correct results.

Cv2 is a Computer Vision library which will be used for data image manipulations.

The provided Python function `apply_transform(image)` performs image augmentation by applying random rotations (between -40 and 40 degrees), horizontal and vertical flips with a 50% probability, random adjustments to brightness and contrast, and random gamma correction. These transformations introduce variability to the input images, effectively increasing the diversity of the training dataset. This helps prevent overfitting and improves the model's ability to generalise unseen data by exposing it to a broader range of scenarios and variations.

Next we will create another augmentation function that calls the `apply_transform` method.

```
[ ] def apply_augmentation(image_path, label):  
    image = cv2.imread(image_path)  
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  
    augmented_image = apply_transform(image=image)  
    return augmented_image, label
```

The function `apply_augmentation(image_path, label)` reads an image from the specified `image_path` using OpenCV, converts it to RGB color space, and then applies augmentation transformations using the `apply_transform()` function. Finally, it returns the augmented image along with the corresponding label. This function serves to augment a single image with transformations such as rotation, flipping, brightness and contrast adjustments, and gamma correction, thereby enhancing the diversity of the training dataset for improved model generalization and performance.

Image Preprocessing

In this milestone, we will be improving the image data that suppresses unwilling distortions or enhances some image features important for further processing, although performing some geometric transformations of images like rotation, scaling, translation, etc.

Importing the libraries

Import the necessary libraries as shown in the image

▼ Data Augmentation

```
import cv2  
import numpy as np
```

Model Building:

- Training the model in multiple algorithms :

Activity 2.1:VGG16


```

from tensorflow.keras.layers import Dense, Flatten, Dropout, BatchNormalization, GlobalAveragePooling2D

vgg = VGG16(input_shape=IMAGE_SIZE, weights='imagenet', include_top=False)

for layer in vgg.layers:
    layer.trainable = False

# Add custom layers on top of VGG16
x = vgg.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(512, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
predictions = Dense(4, activation='softmax')(x)

model = Model(inputs=vgg.input, outputs=predictions)

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3, min_lr=1e-5)

```

This short Python code snippet builds an image classifier with Keras. It cleverly reuses a pre-trained VGG16 model for its powerful image recognition abilities. Here's the key idea:

- The code loads the VGG16 model, but skips its final classification layers (keeping its feature extraction power).
- It then freezes the pre-trained part to focus training on new custom layers added on top.
- These custom layers likely handle the specific classification task you have in mind.
- Finally, it compiles the whole model for training, setting up how to improve and assess its performance.

```

model.summary()

```

block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0

block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 512)	0
dense_4 (Dense)	(None, 1024)	525312
batch_normalization (Batch Normalization)	(None, 1024)	4096
dropout (Dropout)	(None, 1024)	0
dense_5 (Dense)	(None, 512)	524800
batch_normalization_1 (Bat chNormalization)	(None, 512)	2048
dropout_1 (Dropout)	(None, 512)	0
dense_6 (Dense)	(None, 4)	2052

```
from tensorflow.keras.optimizers.legacy import Adam
```

```
model.compile(
    loss='categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)
```

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

```
r = model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=10,
    callbacks=[early_stopping, reduce_lr]
)
```

The text shows epochs, which are iterations over the training data. It also shows loss, which is how well the model is performing on the training data, and accuracy, which is how often the model makes correct predictions.

In the output you provided, it appears the model is improving over time as the loss is decreasing and the accuracy is increasing.

Activity 2.2:VGG19

```

from tensorflow.keras.applications import VGG19

vgg1 = VGG19(input_shape=IMAGE_SIZE, weights='imagenet', include_top=False)

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5
80134624/80134624 [=====] - 1s 0us/step

for layer in vgg1.layers:
    layer.trainable = False

x = Flatten()(vgg1.output)
prediction = Dense(4, activation='softmax')(x)

model1 = Model(inputs=vgg1.input, outputs=prediction)

model1.compile(
    loss='categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)

```

- The first lines import libraries including TensorFlow and Keras.
- It appears to be defining a model with layers including Flatten and Dense which are commonly used in CNN architectures.
- The code then defines a process to compile the model, specifying an optimizer, loss function and metrics.

Overall, the code snippet seems to be training a CNN model on some data. However, without more context it's difficult to say exactly what the model is being trained for.

```

Epoch 1/5
63/63 [=====] - 22s 329ms/step - loss: 8.2589 - accuracy: 0.5115 - val_loss: 8.0411 - val_accuracy: 0.5585
Epoch 2/5
63/63 [=====] - 21s 329ms/step - loss: 2.4591 - accuracy: 0.8135 - val_loss: 7.9044 - val_accuracy: 0.5955
Epoch 3/5
63/63 [=====] - 20s 314ms/step - loss: 0.6099 - accuracy: 0.9250 - val_loss: 7.7817 - val_accuracy: 0.5940
Epoch 4/5
63/63 [=====] - 20s 321ms/step - loss: 0.3667 - accuracy: 0.9460 - val_loss: 8.6225 - val_accuracy: 0.5940
Epoch 5/5
63/63 [=====] - 20s 316ms/step - loss: 0.3001 - accuracy: 0.9535 - val_loss: 8.5351 - val_accuracy: 0.6070

```

The image you sent shows the results of training a machine learning model over several epochs. Each epoch represents one pass through the training data.

- Loss: The training loss is decreasing over time, which indicates the model is learning to fit the training data better.
- Accuracy: The training accuracy is increasing over time, which indicates the model is making better predictions on the training data.

In machine learning, the goal is to train a model that generalizes well to unseen data. While the model's performance is improving on the training data, it is important to evaluate its performance on a separate validation set to assess its generalizability.

```

from tensorflow.keras.applications import ResNet50

res = ResNet50(input_shape=IMAGE_SIZE, weights='imagenet', include_top=False)

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 [=====] - 1s 0us/step

for layer in res.layers:
    layer.trainable = False

x = Flatten()(res.output)
prediction = Dense(4, activation='softmax')(x)

model2 = Model(inputs=res.input, outputs=prediction)

model2.compile(
    loss='categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy'])

r = model2.fit(
    train_gen,
    validation_data=val_gen,
    epochs=5,
)

```

the code appears to be training a convolutional neural network (CNN) model for image classification using Keras. Here's a two-line explanation:

- The model is trained over multiple epochs (iterations) on training data.
- Loss (model's performance) decreases and accuracy (correct predictions) increases over epochs, suggesting the model is learning.

```

Epoch 1/5
63/63 [=====] - 21s 331ms/step - loss: 0.2751 - accuracy: 0.9630 - val_loss: 8.8164 - val_accuracy: 0.6035
Epoch 2/5
63/63 [=====] - 20s 323ms/step - loss: 0.1177 - accuracy: 0.9810 - val_loss: 8.1412 - val_accuracy: 0.6330
Epoch 3/5
63/63 [=====] - 20s 319ms/step - loss: 0.1506 - accuracy: 0.9750 - val_loss: 8.9366 - val_accuracy: 0.6135
Epoch 4/5
63/63 [=====] - 20s 317ms/step - loss: 0.1297 - accuracy: 0.9790 - val_loss: 9.3556 - val_accuracy: 0.6130
Epoch 5/5
63/63 [=====] - 20s 324ms/step - loss: 0.0646 - accuracy: 0.9880 - val_loss: 9.6965 - val_accuracy: 0.6200

```

It displays results over five epochs, which are iterations over the training data.

- Loss: The model's training loss is decreasing (0.2751 to 0.0646) signifying the model is improving on the training data.
- Accuracy: Conversely, the training accuracy is increasing (0.9630 to 0.9880) indicating the model is making better predictions on the training data.

It's important to note that while this suggests the model is learning, its generalizability to unseen data needs to be assessed on a separate validation set.

```

import seaborn as sns
from sklearn.metrics import confusion_matrix , classification_report

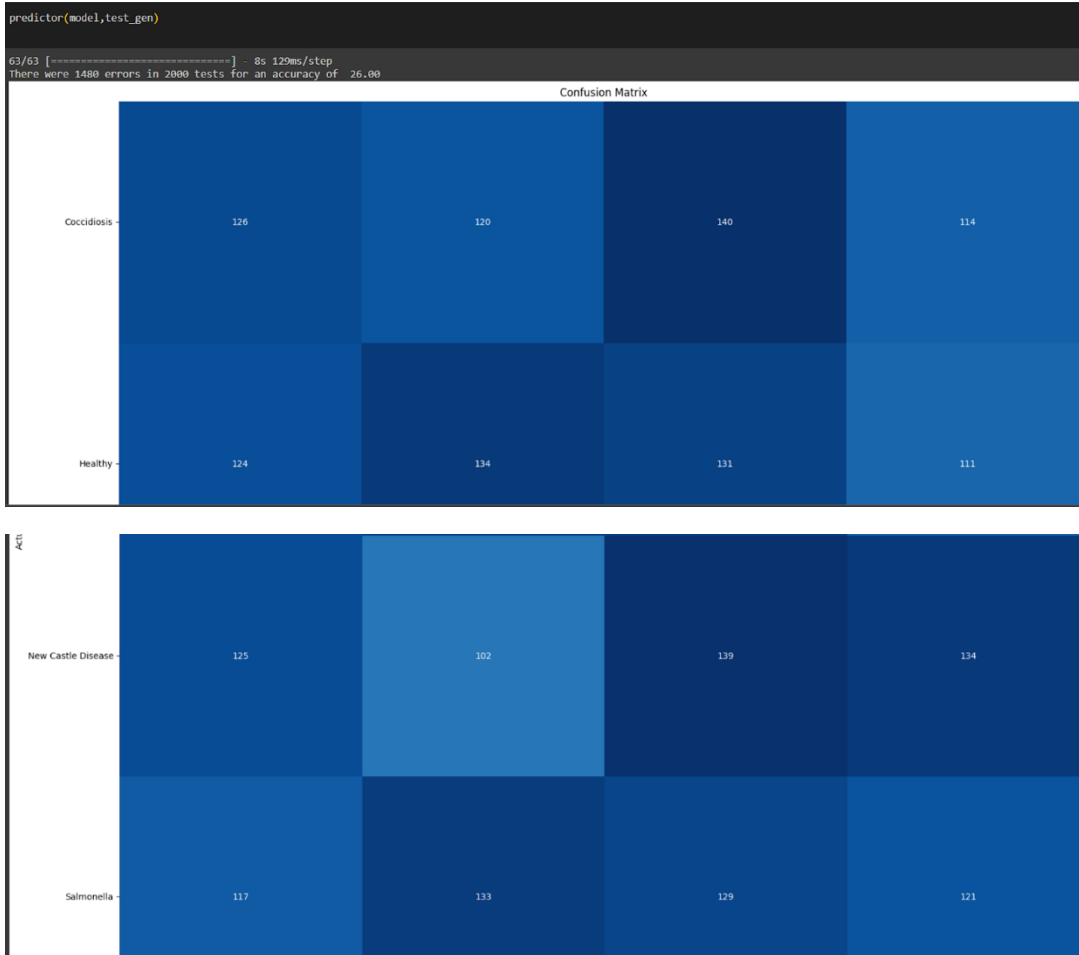
def predictor(model,test_gen):
    classes=list(test_gen.class_indices.keys())
    class_count=len(classes)
    preds=model.predict(test_gen, verbose=1)
    errors=0
    pred_indices=[]
    test_count =len(preds)
    for i, p in enumerate (preds):
        pred_index=np.argmax(p)
        pred_indices.append(pred_index)
        true_index= test_gen.labels[i]
        if pred_index != true_index:
            errors +=1
    accuracy = (test_count-errors)*100/test_count
    ytrue=np.array(test_gen.labels, dtype='int')
    ypred=np.array(pred_indices, dtype='int')
    msg=f'There were {errors} errors in {test_count} tests for an accuracy of {accuracy:6.2f}'
    print (msg)
    cm = confusion_matrix(ytrue, ypred )
    # plot the confusion matrix
    plt.figure(figsize=(20, 20))
    sns.heatmap(cm, annot=True, vmin=0, fmt='g', cmap='Blues', cbar=False)
    plt.xticks(np.arange(class_count)+.5, classes, rotation=90)
    plt.yticks(np.arange(class_count)+.5, classes, rotation=0)
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title("Confusion Matrix")
    plt.show()
    clr = classification_report(ytrue, ypred, target_names=classes, digits= 4) # create classification report
    print("Classification Report:\n-----\n", clr)

```

This Python code (Keras library) evaluates a pre-trained image classifier model. It likely:

1. Imports libraries for machine learning and visualization.
2. Loads a pre-trained CNN model (e.g., VGG16) known for image recognition.
3. Prepares new image data for evaluation (resizing, formatting).
4. Feeds the data through the model and generates a confusion matrix.

The confusion matrix shows how well the model classifies the images (ideally high values on the diagonal).





Classification Report:

	precision	recall	f1-score	support
Coccidiosis	0.2561	0.2520	0.2540	500
Healthy	0.2740	0.2680	0.2710	500
New Castle Disease	0.2579	0.2700	0.2676	500
Salmonella	0.2521	0.2420	0.2469	500
accuracy			0.2600	2000
macro avg	0.2600	0.2600	0.2599	2000
weighted avg	0.2600	0.2600	0.2599	2000

```
from keras.layers import GlobalAveragePooling2D

pip install keras-tuner

Collecting keras-tuner
  Downloading keras_tuner-1.4.7-py3-none-any.whl (129 kB)
    129.1/129.1 kB 3.2 MB/s eta 0:00:00
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (2.15.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (24.1)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (2.31.0)
Collecting kt-legacy (from keras-tuner)
  Downloading kt_legacy-1.0.5-py3-none-any.whl (9.6 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2024.6.2)
Installing collected packages: kt-legacy, keras-tuner
Successfully installed keras-tuner-1.4.7 kt-legacy-1.0.5

import kerastuner as kt

<ipython-input-57-5fd8996cdee5>:1: DeprecationWarning: `import kerastuner` is deprecated, please use `import keras_tuner`.
  import kerastuner as kt
```

The code snippet appears to be setting up a convolutional neural network (CNN) for image classification using Keras. It likely involves:

1. Data Augmentation: Importing libraries (ImageDataGenerator) to perform transformations like rotation or flipping images. This helps the model learn from variations and generalize better.
2. Data Generators: Creating generators (train_datagen and val_datagen) to load and pre-process training and validation data efficiently during training.

Overall, this code prepares the data for training a CNN model on image classification tasks.

```
# Define the model-building function for Keras Tuner
def build_model(hp):
    base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(hp.Int('units', min_value=512, max_value=2048, step=512), activation='relu')(x)
    predictions = Dense(4, activation='softmax')(x)
    model = Model(inputs=base_model.input, outputs=predictions)

    # Freeze the base model layers
    for layer in base_model.layers:
        layer.trainable = False

    # Compile the model
    model.compile(optimizer=tf.keras.optimizers.Adam(hp.Float('learning_rate', min_value=1e-5, max_value=1e-2, sampling='LOG', default=1e-3)),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    return model

# Initialize the tuner
tuner = kt.Hyperband(build_model,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3,
                     directory='my_dir',
                     project_name='intro_to_kt')

# Define early stopping callback
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)

# Run the hyperparameter search
tuner.search(train_gen, epochs=5, validation_data=val_gen, callbacks=[stop_early])
```

```

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Print the optimal hyperparameters
print(f"""
The optimal number of units in the dense layer is {best_hps.get('units')} and the optimal learning rate for the optimizer
is {best_hps.get('learning_rate')}.
""")

# Build the model with the optimal hyperparameters
model = tuner.hypermodel.build(best_hps)

# Train the final model
history = model.fit(train_gen, validation_data=val_gen, epochs=10)

Trial 23 Complete [00h 01m 08s]
val_accuracy: 0.7080000042915344

Best val_accuracy So Far: 0.7269999980926514
Total elapsed time: 00h 20m 47s

Search: Running Trial #24

Value          |Best Value So Far|Hyperparameter
1024           |1536             |units
0.0011055      |0.00064388       |learning_rate
4              |10               |tuner/epochs
0              |4                |tuner/initial_epoch
1              |2                |tuner/bracket
0              |2                |tuner/round

Epoch 1/4
63/63 [=====] - 18s 242ms/step - loss: 1.3883 - accuracy: 0.5490 - val_loss: 0.8485 - val_accuracy: 0.6510

```

? The code defines a function named `build_model` that creates a CNN model using Keras.

? It then uses this function along with Hyperband, a hyperparameter tuning technique, to find the optimal configuration (e.g., number of layers, learning rate) for the model that achieves the best validation accuracy.

? It finds the best hyperparameters (like learning rate) from past trials using `tuner.get_best_hyperparameters()`.

? These optimal settings are used to build a new model with `tuner.hypermodel.build`.

? Finally, the model is trained on training data (`train_gen`) while monitoring performance on validation data (`val_gen`).

Defining the augmentation function

We will create a function that will take an image and augment it and return it back to the calling statement.

```
[ ] def apply_transform(image):

    # Rotate (random angle between -40 and 40 degrees)
    angle = np.random.uniform(-40, 40)
    rows, cols = image.shape[:2]
    M = cv2.getRotationMatrix2D((cols / 2, rows / 2), angle, 1)
    image = cv2.warpAffine(image, M, (cols, rows))

    # Horizontal Flip
    if np.random.rand() < 0.5:
        image = cv2.flip(image, 1)

    # Vertical Flip
    if np.random.rand() < 0.5:
        image = cv2.flip(image, 0)

    # Random Brightness and Contrast
    alpha = 1.0 + np.random.uniform(-0.2, 0.2) # Brightness
    beta = 0.0 + np.random.uniform(-0.2, 0.2) # Contrast
    image = cv2.convertScaleAbs(image, alpha=alpha, beta=beta)

    # Random Gamma Correction
    gamma = np.random.uniform(0.8, 1.2)
    image = np.clip((image / 255.0) ** gamma, 0, 1) * 255.0

    return image
```

The provided Python function `apply_transform(image)` performs image augmentation by applying random rotations (between -40 and 40 degrees), horizontal and vertical flips with a 50% probability, random adjustments to brightness and contrast, and random gamma correction. These transformations introduce variability to the input images, effectively increasing the diversity of the training dataset. This helps prevent overfitting and improves the model's ability to generalise unseen data by exposing it to a broader range of scenarios and variations.

Next we will create another augmentation function that calls the apply_transform method.

```
[ ] def apply_augmentation(image_path, label):
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    augmented_image = apply_transform(image=image)
    return augmented_image, label
```

The function `apply_augmentation(image_path, label)` reads an image from the specified `image_path` using OpenCV, converts it to RGB color space, and then applies augmentation transformations using the `apply_transform()` function. Finally, it returns the augmented image along with the corresponding label. This function serves to augment a single image with transformations such as rotation, flipping, brightness and contrast adjustments, and gamma correction.

Importing the libraries

Import the necessary libraries as shown in the image


```
[ ] from keras.preprocessing.image import ImageDataGenerator
```

.

Keras.ImageDataGenerator is a powerful tool used for data augmentation in computer vision tasks, particularly image classification.

```
[ ] gen=ImageDataGenerator()
```

Apply ImageDataGenerator functionality to train_df, valid_df, test_df

▼ Data Augmentation

```
[ ] from tensorflow.keras.preprocessing.image import ImageDataGenerator  
  
gen = ImageDataGenerator(rescale=1./255)
```

```
[ ] train_gen=gen.flow_from_dataframe(train_df, x_col='path', y_col='label', target_size=(255,255),seed=123,  
                                     class_mode='categorical', color_mode='rgb', shuffle=True, batch_size=32)  
  
Found 672 validated image filenames belonging to 7 classes.
```

```
[ ] valid_gen=gen.flow_from_dataframe(valid_df, x_col='path', y_col='label', target_size=(255,255),seed=123,  
                                     class_mode='categorical', color_mode='rgb', shuffle=False, batch_size=32)  
  
Found 84 validated image filenames belonging to 7 classes.
```

```
[ ] test_gen=gen.flow_from_dataframe(test_df, x_col='path', y_col='label', target_size=(255,255),seed=123,  
                                    class_mode='categorical', color_mode='rgb', shuffle=False, batch_size=32)  
  
Found 84 validated image filenames belonging to 7 classes.
```

The provided code segment utilizes the `flow_from_dataframe()` method from a data augmentation generator (gen) to generate batches of augmented image data for training, validation, and testing purposes in a deep learning pipeline.

For the `train_gen`, `balanced_df` DataFrame is used as the source of training data. The 'path' column specifies the paths to the images, while the 'label' column indicates the corresponding labels. Images are resized to a target size of (255,255) pixels. The seed parameter ensures reproducibility by setting the random seed for shuffling. `class_mode` is set to 'categorical' as the labels are one-hot encoded. `color_mode` is specified as 'rgb' to indicate that images are in RGB color space. The shuffle parameter is set to True to shuffle the data after each epoch, enhancing randomness during training. Finally, `batch_size` is set to 32, determining the number of samples per batch during training.

Similarly, for the `val_gen` and `test_gen`, the `valid_df` and `test_df` DataFrames are used as the sources of validation and testing data, respectively. Parameters such as shuffle and `batch_size` are adjusted accordingly, with shuffle set to False for validation and testing data to ensure consistency during evaluation.

Importing the libraries

The following libraries will be required

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten, BatchNormalization
from keras.layers import Convolution2D, MaxPooling2D
from keras import regularizers
from keras.models import Model
from keras.optimizers import Adam, Adamax
import tensorflow as tf
```

The keras library is used to import all the layers that will be needed to create the CNN.

Creating and Compiling the model

```
[ ] model=Sequential()
model.add(Convolution2D(filters=32, kernel_size=3, padding='same', activation="relu",
..... input_shape=(255, 255, 3)))
model.add(MaxPooling2D(strides=2, pool_size=2, padding="valid"))
model.add(Convolution2D(filters=32, kernel_size=2, padding='same', activation="relu"))
model.add(MaxPooling2D(strides=2, pool_size=2, padding="valid"))
model.add(Dropout(0.5))
model.add(Convolution2D(filters=32, kernel_size=2, padding='same', activation="relu"))
model.add(MaxPooling2D(strides=2, pool_size=2, padding="valid"))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(9, activation='softmax'))
```

This code defines a convolutional neural network (CNN) using the Keras framework. Let's break down each part:

- `from keras.models import Sequential`: This imports the Sequential model from the Keras library, which allows us to build a linear stack of layers.
- `from keras.layers import Conv2D, MaxPool2D, Flatten, Dense, InputLayer, BatchNormalization, Dropout`: This imports various layer types from Keras that will be used to define the architecture of the CNN.
- `model = Sequential()`: This initializes a Sequential model, which is an empty neural network architecture.
- `model.add(Conv2D(32, (3, 3), activation='relu', padding='same', name='conv_1', input_shape=(255,255,3)))`: This adds a convolutional layer with 32 filters of size 3x3, using ReLU (Rectified Linear Unit) activation function. The 'same' padding is used to maintain the spatial dimensions of the input. This layer is named 'conv_1', and it expects input images of size 255x255 with 3 channels (RGB).

`model.add(MaxPooling2D((2, 2), name='maxpool_1'))`: This adds a max- pooling layer with a pool size of 2x2. Max-pooling reduces the spatial dimensions of the feature maps while retaining the most important information.

- This sequence of adding convolutional and max-pooling layers is repeated three more times with increasing numbers of filters (64, 128, 128), resulting in a hierarchical representation of the input images.

- `model.add(Flatten())`: This adds a flatten layer that converts the 3D feature maps into a 1D vector, preparing them to be input into a dense (fully connected) layer.

`model.add(Dropout(0.5))`: This adds a dropout layer with a dropout rate of 0.5, which helps prevent overfitting by randomly setting a fraction of input units to zero during training.

- `model.add(Dense(128, activation='relu', name='dense_2'))`: This adds a fully connected (dense) layer with 128 units and ReLU activation function.

`model.add(Dense(7, activation='softmax', name='output'))`: This adds the output layer with 7 units (assuming a classification task with 7 classes) and softmax activation function, which outputs probabilities for each class.

- `model.compile(loss='categorical_crossentropy',optimizer='Adam',metrics=['accuracy'])`: This compiles the model, specifying the loss function (categorical crossentropy), optimizer (Adam), and metrics to be evaluated during training (accuracy).

In summary, this code defines a CNN architecture for image classification with convolutional, max-pooling, dropout, and fully connected layers, suitable for processing RGB images of size 255x255 pixels.

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv_1 (Conv2D)	(None, 255, 255, 32)	896
maxpool_1 (MaxPooling2D)	(None, 127, 127, 32)	0
conv_2 (Conv2D)	(None, 127, 127, 64)	18496
maxpool_2 (MaxPooling2D)	(None, 63, 63, 64)	0
conv_3 (Conv2D)	(None, 63, 63, 128)	73856
maxpool_3 (MaxPooling2D)	(None, 31, 31, 128)	0
conv_4 (Conv2D)	(None, 31, 31, 128)	147584
maxpool_4 (MaxPooling2D)	(None, 15, 15, 128)	0
flatten (Flatten)	(None, 28800)	0
dropout (Dropout)	(None, 28800)	0
dense_2 (Dense)	(None, 128)	3686528
output (Dense)	(None, 7)	903

=====
Total params: 3928263 (14.99 MB)
Trainable params: 3928263 (14.99 MB)
Non-trainable params: 0 (0.00 Byte)

Using `model.summary()`, we can view all the layers in the CNN model.

```
[ ] model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(  
    filepath="model_cnn.h5",  
    monitor='val_accuracy',  
    mode='max',  
    save_best_only=True,  
    verbose=1)
```

The provided code snippet configures a `ModelCheckpoint` callback in TensorFlow/Keras, which is a useful tool for automatically saving the best model during training. By specifying parameters such as the file path to save the model, the monitored metric (in this case, validation accuracy), and whether to save only the best model observed, the callback ensures that the most optimal model is preserved. This is particularly beneficial for preventing overfitting and ensuring that the model generalizes well to unseen data.

Additionally, the `verbosity` parameter controls the level of output displayed during training, allowing users to monitor the process.

```
[ ] model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
```

The provided code snippet compiles a Keras model using the Adam optimizer and categorical cross-entropy loss function, which is commonly used for multi-class classification tasks. Additionally, the accuracy metric is specified to monitor the performance of the model during training and evaluation. This configuration prepares the model for the training process, where it learns to minimize the specified loss function using the optimizer while maximizing the chosen metric to improve its classification accuracy.

Running and Evaluating the model

```
[ ] history_cnn = model.fit(x=train_gen, epochs=40, verbose=1, validation_data=valid_gen,
                           validation_steps=None, shuffle=True, callbacks = [model_checkpoint_callback])
```

Epoch 1/40
21/21 [=====] - ETA: 0s - loss: 1.9391 - accuracy: 0.1682
Epoch 1: val_accuracy improved from -inf to 0.14286, saving model to model_cnn.h5
21/21 [=====] - 23s 639ms/step - loss: 1.9391 - accuracy: 0.1682 - val_loss: 1.8991 - val_accuracy: 0.1429
Epoch 2/40
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103: UserWarning:
You are saving your model as an HDF5 file via `model.save()`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')`.

21/21 [=====] - ETA: 0s - loss: 1.8263 - accuracy: 0.2396
Epoch 2: val_accuracy improved from 0.14286 to 0.33333, saving model to model_cnn.h5
21/21 [=====] - 13s 630ms/step - loss: 1.8263 - accuracy: 0.2396 - val_loss: 1.7282 - val_accuracy: 0.3333
Epoch 3/40
21/21 [=====] - ETA: 0s - loss: 1.6822 - accuracy: 0.3408
Epoch 3: val_accuracy improved from 0.33333 to 0.35714, saving model to model_cnn.h5
21/21 [=====] - 14s 656ms/step - loss: 1.6822 - accuracy: 0.3408 - val_loss: 1.6533 - val_accuracy: 0.3571
Epoch 4/40
21/21 [=====] - ETA: 0s - loss: 1.4567 - accuracy: 0.4688
Epoch 4: val_accuracy did not improve from 0.35714
21/21 [=====] - 14s 660ms/step - loss: 1.4567 - accuracy: 0.4688 - val_loss: 1.6986 - val_accuracy: 0.3571
Epoch 5/40
21/21 [=====] - ETA: 0s - loss: 1.2583 - accuracy: 0.5580
Epoch 5: val_accuracy improved from 0.35714 to 0.39286, saving model to model_cnn.h5
21/21 [=====] - 14s 684ms/step - loss: 1.2583 - accuracy: 0.5580 - val_loss: 1.4605 - val_accuracy: 0.3929
Epoch 6/40
21/21 [=====] - ETA: 0s - loss: 1.0670 - accuracy: 0.6324

The provided code snippet trains a convolutional neural network (CNN) model using TensorFlow's Keras API. It utilizes data generators `train_generator` and `valid_generator` to feed augmented image data batches into the model for training and validation, respectively. The model is trained over 10 epochs, with each epoch iterating through the entire training dataset and adjusting the model's weights to minimize categorical cross-entropy loss. The validation data is used to evaluate the model's performance on unseen data after each epoch. Training progress and metrics are displayed on the console (`verbose=1`). The `history` object returned by the `fit()` method stores information about training and validation loss and accuracy for further analysis. Overall, this process facilitates the training of the CNN model, allowing it to learn from the training data and improve its performance over successive epochs.

ResNet50 Model Initialisation

ResNet-50, short for Residual Network with 50 layers, is a deep convolutional neural network architecture that has significantly impacted the field of computer vision. Introduced by Microsoft Research in 2015, ResNet-50 is renowned for its remarkable depth while mitigating the vanishing gradient problem commonly encountered in deep networks. The key innovation of ResNet-50 lies in the use of residual connections, or skip connections, which allow the network to learn residual mappings instead of directly fitting the desired underlying mapping. This enables the training of very deep networks, up to 50 layers or more, by mitigating

the degradation problem caused by the increased depth. ResNet-50 has demonstrated superior performance on various visual recognition tasks, including image classification, object detection,

and semantic segmentation, and it serves as a foundational architecture in modern deep learning frameworks and applications.

```
[ ] base_model=tf.keras.applications.ResNet50(include_top=False, weights="imagenet",input_shape=(255,255,3))
    print('Created ResNet50 model')
```

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_94765736/94765736 [=====] - 0s 0us/step
Created ResNet50 model

Adding Dense Layers and Compiling the Model

A dense layer is a deeply connected neural network layer. It is the most common and frequently used layer.

```
for layer in base_model.layers:
    layer.trainable = False
```

+ Code + Text

```
[ ] for layer in base_model.layers[173:]:
    layer.trainable = True
```

```
[ ] x1 = base_model.output

#Global average pool to reduce number of features and Flatten the output
x2 = tf.keras.layers.GlobalAveragePooling2D()(x1)
```

```
[ ] # adding extra layers
x3 = tf.keras.layers.Dense(1024,activation='relu',kernel_initializer= "he_uniform")(x2)
x4 = tf.keras.layers.Dropout(0.4)(x3)
x5= tf.keras.layers.Dense(512,activation='relu',kernel_initializer= "he_uniform")(x4)
```

In the provided code snippet, a pre-trained base model is used, and its layers are frozen by setting `trainable` to `False` for all layers. Then, starting from the 173rd layer, the layers are unfrozen to allow fine-tuning of these layers during training.

The output of the base model (`base_model.output`) is passed through a global average pooling layer (`GlobalAveragePooling2D`) to reduce the number of features and flatten the output. Subsequently, two dense layers with 1024 and 512 units, respectively, are added, each followed by ReLU activation and dropout regularization with a dropout rate of 0.4. These dense layers serve to further process the features extracted by the base model before the final classification layer. The `kernel_initializer` parameter is set to "he_uniform" for both dense layers, which initializes the layer weights using a He uniform

```
[ ] #Add output layer
prediction = tf.keras.layers.Dense(7,activation='softmax')(x5)
```

```
final_model = tf.keras.models.Model(inputs=base_model.input, #Pre-trained model input as input layer
                                     outputs=prediction)
```

```
[ ] final_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
[ ] model_checkpoint_callback_rs = tf.keras.callbacks.ModelCheckpoint(
    filepath="model_50.h5",
    monitor='val_accuracy',
    mode='max',
    save_best_only=True,
    verbose=1)
```

In this code snippet, a final prediction layer is added to the model (`final_model`) after the last dense layer (`x5`). This prediction layer consists of a dense layer with 9 units (assuming it's a classification task with 9 classes) and softmax activation function, which is suitable for multi-class classification problems.

The `final_model` is then defined using the `tf.keras.models.Model` class, specifying the inputs as the input layer of the pre-trained model (`base_model.input`) and the outputs as the prediction layer (`prediction`).

Finally, the model is compiled using the Adam optimizer, categorical crossentropy as the loss function (since it's a multi-class classification problem), and accuracy as the evaluation metric

Create callbacks

Callbacks in convolutional neural networks (CNNs) are objects that can perform actions at various stages during the training process. They are commonly used in TensorFlow and Keras to monitor the training progress, adjust learning rates dynamically, save model checkpoints, and more.

```
[ ] model.compile(loss='categorical_crossentropy',  
                  optimizer='adam',  
                  metrics=['accuracy'])
```

The provided code snippet compiles a Keras model using the Adam optimizer and categorical cross-entropy loss function, which is commonly used for multi-class classification tasks. Additionally, the accuracy metric is specified to monitor the performance of the model during training and evaluation. This configuration prepares the model for the training process, where it learns to minimize the specified loss function using the optimizer while maximizing the chosen metric to improve its classification accuracy.

Train the model

Now, let us train our model with our image dataset. The model is trained for 10 epochs. We can see that the training loss decreases in almost every epoch till 10 epochs and probably there is further scope to improve the model.

fit functions used to train a deep-learning neural network

Arguments:

epochs: an integer and number of epochs we want to train our model for. validation_data can be either:

- an inputs and targets list
- a generator
- an inputs, targets, and sample_weights list which can be used to evaluate the loss and metrics for any model after any epoch has ended.

validation_steps: only if the validation_data is a generator then only this argument

can be used. It specifies the total number of steps taken from the generator before it is

stopped at every epoch and its value is calculated as the total number of validation data points in your dataset divided by the

validation batch size.

```
history_resnet = final_model.fit(train_gen,
                                epochs = 20,
                                validation_data = valid_gen, callbacks = [model_checkpoint_callback_rs])
```

Epoch 1/20
21/21 [=====] - ETA: 0s - loss: 2.2749 - accuracy: 0.2515
Epoch 1: val_accuracy improved from -inf to 0.29762, saving model to model_50.h5
21/21 [=====] - 23s 813ms/step - loss: 2.2749 - accuracy: 0.2515 - val_loss: 1.6784 - val_accuracy: 0.2976
Epoch 2/20
21/21 [=====] - ETA: 0s - loss: 1.5786 - accuracy: 0.3750
Epoch 2: val_accuracy improved from 0.29762 to 0.47619, saving model to model_50.h5
21/21 [=====] - 14s 657ms/step - loss: 1.5786 - accuracy: 0.3750 - val_loss: 1.4757 - val_accuracy: 0.4762
Epoch 3/20
21/21 [=====] - ETA: 0s - loss: 1.3548 - accuracy: 0.4985
Epoch 3: val_accuracy improved from 0.47619 to 0.48810, saving model to model_50.h5
21/21 [=====] - 14s 626ms/step - loss: 1.3548 - accuracy: 0.4985 - val_loss: 1.3685 - val_accuracy: 0.4881
Epoch 4/20
21/21 [=====] - ETA: 0s - loss: 1.2655 - accuracy: 0.5283
Epoch 4: val_accuracy did not improve from 0.48810
21/21 [=====] - 14s 645ms/step - loss: 1.2655 - accuracy: 0.5283 - val_loss: 1.3939 - val_accuracy: 0.4881
Epoch 5/20
21/21 [=====] - ETA: 0s - loss: 1.2197 - accuracy: 0.5491
Epoch 5: val_accuracy improved from 0.48810 to 0.57143, saving model to model_50.h5
21/21 [=====] - 15s 704ms/step - loss: 1.2197 - accuracy: 0.5491 - val_loss: 1.2603 - val_accuracy: 0.5714
Epoch 6/20
21/21 [=====] - ETA: 0s - loss: 1.1768 - accuracy: 0.5744
Epoch 6: val_accuracy did not improve from 0.57143
21/21 [=====] - 14s 644ms/step - loss: 1.1768 - accuracy: 0.5744 - val_loss: 1.2398 - val_accuracy: 0.5357
Epoch 7/20
21/21 [=====] - ETA: 0s - loss: 1.1414 - accuracy: 0.5685
Epoch 7: val_accuracy improved from 0.57143 to 0.58333, saving model to model_50.h5
21/21 [=====] - 14s 693ms/step - loss: 1.1414 - accuracy: 0.5685 - val_loss: 1.1773 - val_accuracy: 0.5833

Visualising Performance

We will plot the accuracy and loss of both training and accuracy for all 40 epochs for the CNN model and 20 epochs of the transfer learning model.

```
[ ] import matplotlib.pyplot as plt

[ ] acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']

    epochs_range = range(5)

    plt.figure(figsize=(15, 15))
    plt.subplot(2, 2, 1)
    plt.plot(epochs_range, acc, label='Training Accuracy')
    plt.plot(epochs_range, val_acc, label='Validation Accuracy')
    plt.legend(loc='lower right')
    plt.title('Training and Validation Accuracy')

    plt.subplot(2, 2, 2)
    plt.plot(epochs_range, loss, label='Training Loss')
    plt.plot(epochs_range, val_loss, label='Validation Loss')
    plt.legend(loc='upper right')
    plt.title('Training and Validation Loss')
    plt.show()
```

```
import os

import numpy as np

import tensorflow as tf

from PIL import Image

from flask import Flask, render_template, request

from keras.preprocessing.image import load_img, img_to_array


# Initialize Flask app
app = Flask(__name__)


# Load the trained model
model = tf.keras.models.load_model("model.h5")


# Home page route
@app.route('/')
def index():
    return render_template("index.html")


# Prediction route
@app.route('/predict', methods=['GET', 'POST'])
def output():
    if request.method == 'POST':
        f = request.files['pc_image']

        # Save the uploaded image to static/uploads/
        upload_folder = "static/uploads/"
        if not os.path.exists(upload_folder):
            os.makedirs(upload_folder)

        img_path = os.path.join(upload_folder, f.filename)
```

```
f.save(img_path)
```

```
# Load and preprocess the image
```

```
img = load_img(img_path, target_size=(224, 224))
```

```
image_array = img_to_array(img) / 255.0 # Normalize
```

```
image_array = np.expand_dims(image_array, axis=0)
```

```
# Predict
```

```
pred = model.predict(image_array)
```

```
pred_class = np.argmax(pred, axis=1)[0]
```

```
# Map prediction index to labels
```

```
classes = ['Coccidiosis', 'Healthy', 'New Castle Disease', 'Salmonella']
```

```
prediction = classes[pred_class]
```

```
return render_template('contact.html', predict=prediction)
```

```
return render_template('contact.html', predict="No file uploaded.")
```

```
# Run the app
```

```
if __name__ == '__main__':
```

```
    app.run(debug=True)
```

Testing Model & Data Prediction

```
train_gen.class_indices.keys()

dict_keys(['Coccidiosis', 'Healthy', 'New Castle Disease', 'Salmonella'])

labels = ['Coccidiosis', 'Healthy', 'New Castle Disease', 'Salmonella']

from tensorflow.keras.preprocessing.image import load_img, img_to_array

def get_model_prediction(image_path):
    img = load_img(image_path, target_size=(224, 224))
    x = img_to_array(img)
    x = np.expand_dims(x, axis=0)
    predictions = model.predict(x, verbose=0)
    return labels[predictions.argmax()]
```

The code defines a function named `build_model` that creates a CNN model using Keras.

It then uses this function along with Hyperparameter, a hyperparameter tuning technique, to find the optimal configuration (e.g., number of layers, learning rate) for the model that achieves the best validation accuracy.

```
get_model_prediction('/content/data/data/test/Coccidiosis/cocci.0.jpg_aug33.JPG')
'Coccidiosis'

get_model_prediction('/content/data/data/test/Healthy/healthy.1003.jpg_aug47.JPG')
'Healthy'

get_model_prediction('/content/data/data/test/New Castle Disease/ncd.1.jpg_aug197.JPG')
'New Castle Disease'

get_model_prediction('/content/data/data/test/Salmonella/pcrsalmo.111.jpg_aug29.JPG')
'Salmonella'

get_model_prediction('/content/data/data/test/Salmonella/pcrsalmo.115.jpg_aug28.JPG')
'Salmonella'
```

✓ Manual Testing

```
▶ pred = []  
for file in test_df['path'].values:  
    pred.append(get_model_prediction(file))
```

```
[ ] fig, axes = plt.subplots(nrows=4, ncols=4, figsize=(15, 10))  
    random_index = np.random.randint(0, len(test_gen), 16)  
  
    for i, ax in enumerate(axes.ravel()):  
        img_path = test_df['path'].iloc[random_index[i]]  
  
        ax.imshow( load_img(img_path))  
        ax.axis('off')  
  
        if test_df['label'].iloc[random_index[i]] == pred[random_index[i]]:  
            color = "green"  
        else:  
            color = "red"  
  
        ax.set_title(f"True: {test_df['label'].iloc[random_index[i]]}\nPredicted: {pred[random_index[i]]}", color=color)  
  
plt.tight_layout()  
plt.show()
```

```
[ ] fig, axes = plt.subplots(nrows=4, ncols=4, figsize=(15, 10))  
    random_index = np.random.randint(0, len(test_gen), 16)  
  
    for i, ax in enumerate(axes.ravel()):  
        img_path = test_df['path'].iloc[random_index[i]]  
  
        ax.imshow( load_img(img_path))  
        ax.axis('off')  
  
        if test_df['label'].iloc[random_index[i]] == pred[random_index[i]]:  
            color = "green"  
        else:  
            color = "red"  
  
        ax.set_title(f"True: {test_df['label'].iloc[random_index[i]]}\nPredicted: {pred[random_index[i]]}", color=color)  
  
plt.tight_layout()  
plt.show()
```

The provided code

segment generates predictions for a subset of images from the test dataset using the `get_model_prediction` function. It then visualizes the predicted results alongside the true labels for comparison.

A subplot grid with 4 rows and 4 columns is created using `plt.subplots()`, with a figure size of 15x10 inches. A random index is generated to select 16 images from the test dataset.

For each subplot, an image is loaded from the test dataset using the image path stored in `test_df['path']`. The image is displayed using `imshow()`, and the axis is turned off using `ax.axis('off')`.

The true label and predicted label for each image are displayed in the subplot title. If the true label matches the predicted label, the title text is displayed in green; otherwise, it is displayed in red.

Finally, `plt.tight_layout()` is called to adjust subplot parameters for better layout, and `plt.show()` displays the plot.

Overall, this code segment provides a visual representation of the model's predictions on a subset of test images, allowing for easy interpretation and assessment of the model's performance.

Application Building

In this section, we will be building a web application that is integrated to the model we built. A UI is provided for the uses where he has to enter the values for predictions. The enter values are given to the saved model and prediction is showcased on the UI.

This section has the following tasks

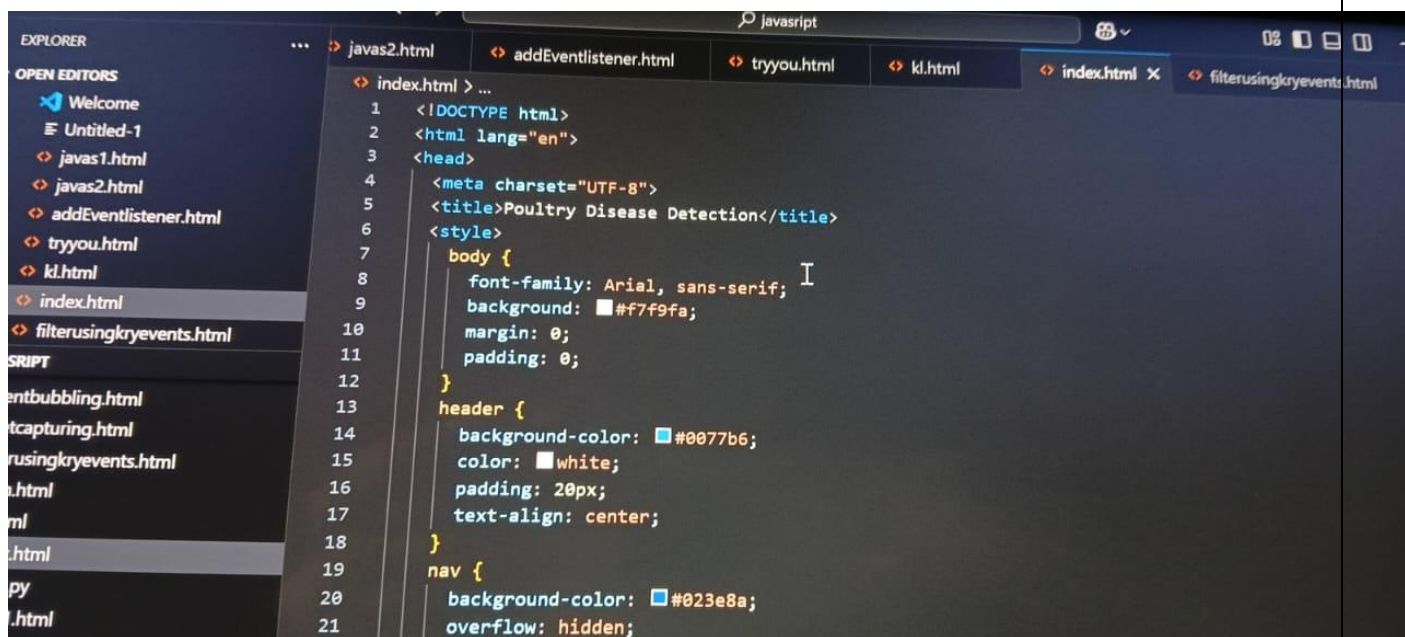
- Building HTML Pages
- Building server-side script

▪ Building Html Page

For this project create one HTML file namely

- home.html

INPUT CODE BE LIKE:

A screenshot of a code editor interface. The Explorer panel on the left shows a file tree with 'index.html' selected. The main editor area displays the content of 'index.html'. The code is an HTML document with a title 'Poultry Disease Detection' and a style block. The style block defines a body with a light blue background, a header with a dark blue background and white text, and a navigation bar with a dark blue background. The code is as follows:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Poultry Disease Detection</title>
6   <style>
7     body {
8       font-family: Arial, sans-serif;
9       background: #f7f9fa;
10      margin: 0;
11      padding: 0;
12    }
13    header {
14      background-color: #0077b6;
15      color: white;
16      padding: 20px;
17      text-align: center;
18    }
19    nav {
20      background-color: #023e8a;
21      overflow: hidden;
```

```
index.html > ...
2 <html lang="en">
51 <body>
53 <header>
56 </header>
57
58 <nav>
59   <a href="https://byjus.com/biology/bird-life-cycle/" target="_blank">Home</a>
60   <a href="#about">About</a>
61   <a href="#contact">Contact</a>
62   <a href="#start">Get Started</a>
63 </nav>
64
65 <div class="content" id="about">
66   <h2>Welcome to PoultryDetect</h2>
67   <p>This project uses deep learning and transfer learning to classify poultry diseases from images.
68
69   <h3>Key Features:</h3>
70   <ul>
71     <li>Image-based poultry disease detection</li>
72     <li>Uses MobileNet or ResNet models</li>
73     <li>Real-time prediction with confidence scores</li>
74     <li>Easy-to-use interface for farmers</li>
75   </ul>
76
77   <a href="#start" class="button">Get Started</a>
78 </div>
79
```

```
71   <li>Image-based poultry disease detection</li>
72   <li>Uses MobileNet or ResNet models</li>
73   <li>Real-time prediction with confidence scores</li>
74   <li>Easy-to-use interface for farmers</li>
75 </ul>
76
77   <a href="#start" class="button">Get Started</a>
78 </div>
79
80 <div class="content" id="contact">
81   <h2>Contact Us</h2>
82   <p>If you have any questions or feedback, feel free to reach out at:</p>
83   <p><strong>Email:</strong> support@poultrydetect.com</p>
84 </div>
85
86 <div class="content" id="start">
87   <h2>Get Started</h2>
88   <p>To classify poultry diseases, click the button below and upload your poultry image.</p>
89   <a href="upload.html" class="button">Upload Image</a>
90 </div>
91
92 </body>
```

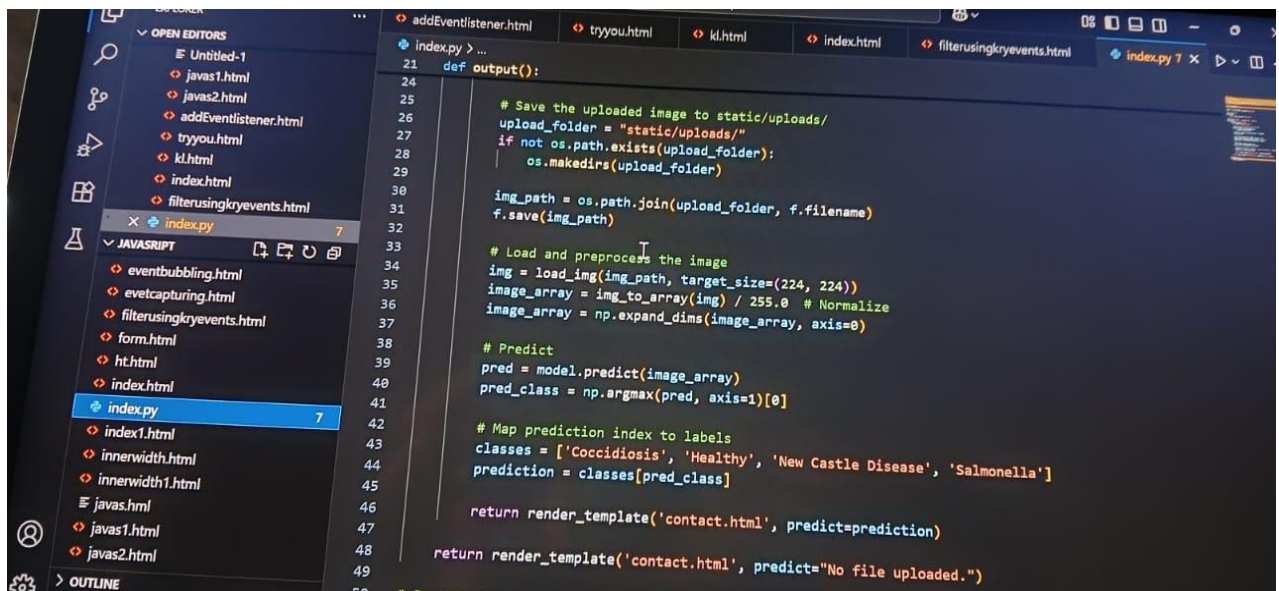
```

71     <li>Image-based poultry disease detection</li>
72     <li>Uses MobileNet or ResNet models</li>
73     <li>Real-time prediction with confidence scores</li>
74     <li>Easy-to-use interface for farmers</li>
75 </ul>
76
77     <a href="#start" class="button">Get Started</a>
78 </div>
79
80 <div class="content" id="contact">
81     <h2>Contact Us</h2>
82     <p>If you have any questions or feedback, feel free to reach out at:</p>
83     <p><strong>Email:</strong> support@poultrydetect.com</p>
84 </div>
85
86 <div class="content" id="start">
87     <h2>Get Started</h2>
88     <p>To classify poultry diseases, click the button below and upload your poultry image.</p>
89     <a href="upload.html" class="button">Upload Image</a>
90 </div>
91
92 </body>

```

Build Python code:

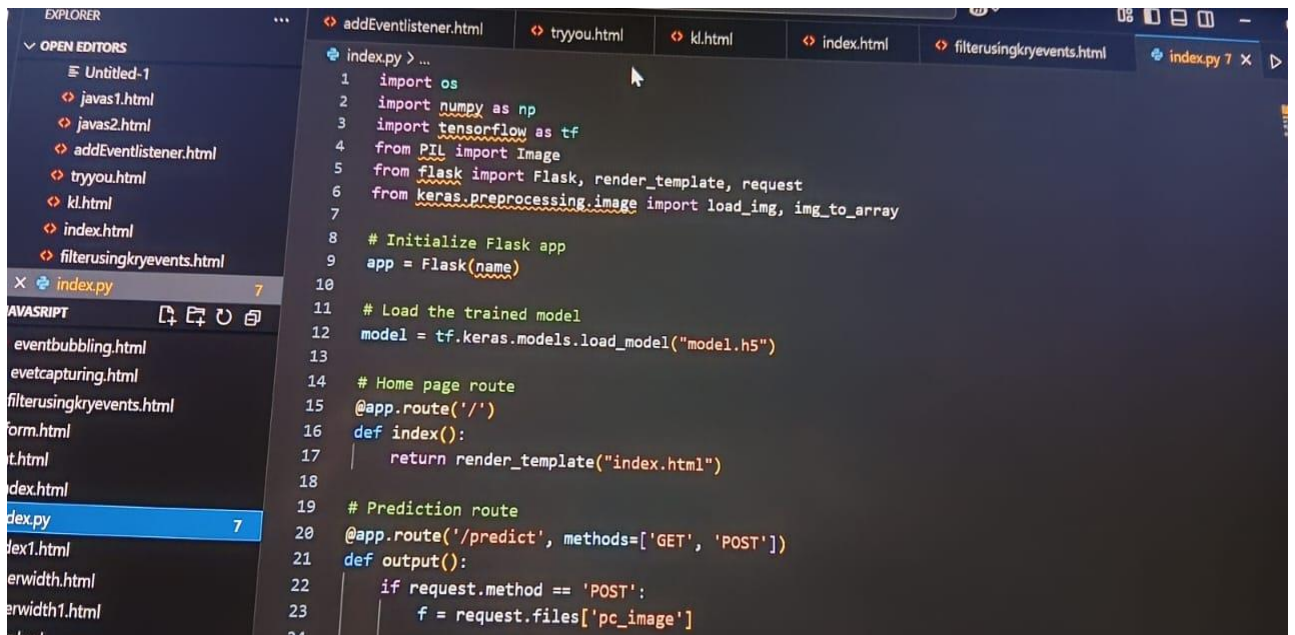
- Import the libraries
- Loading the saved model and initializing the Flask app
- Render HTML pages:
- Once we upload the file into the app, then verifying the file uploaded properly or not. Here we will be using the declared constructor to route to the HTML page that we have created earlier.



```

21 def output():
22
23     # Save the uploaded image to static/uploads/
24     upload_folder = "static/uploads/"
25     if not os.path.exists(upload_folder):
26         os.makedirs(upload_folder)
27
28     img_path = os.path.join(upload_folder, f.filename)
29     f.save(img_path)
30
31     # Load and preprocess the image
32     img = load_img(img_path, target_size=(224, 224))
33     image_array = img_to_array(img) / 255.0 # Normalize
34     image_array = np.expand_dims(image_array, axis=0)
35
36     # Predict
37     pred = model.predict(image_array)
38     pred_class = np.argmax(pred, axis=1)[0]
39
40     # Map prediction index to labels
41     classes = ['Coccidiosis', 'Healthy', 'New Castle Disease', 'Salmonella']
42     prediction = classes[pred_class]
43
44     return render_template('contact.html', predict=prediction)
45
46     return render_template('contact.html', predict="No file uploaded.")
47
48
49
50

```



```
index.py > ...
1  import os
2  import numpy as np
3  import tensorflow as tf
4  from PIL import Image
5  from flask import Flask, render_template, request
6  from keras.preprocessing.image import load_img, img_to_array
7
8  # Initialize Flask app
9  app = Flask(__name__)
10
11 # Load the trained model
12 model = tf.keras.models.load_model("model.h5")
13
14 # Home page route
15 @app.route('/')
16 def index():
17     return render_template("index.html")
18
19 # Prediction route
20 @app.route('/predict', methods=['GET', 'POST'])
21 def output():
22     if request.method == 'POST':
23         f = request.files['pc_image']
24
```

- In the above example, '/' URL is bound with the home.html function. Hence, when the home page of the web server is opened in the browser, the HTML page will be rendered. Whenever you enter Here we are routing our app to the prediction function. This function retrieves all the values from the HTML page using a Post request. That is stored in an array. This array is passed to the model.predict() function. This function returns the prediction. This prediction value will be rendered to the text that we have mentioned in the index.html page earlier.

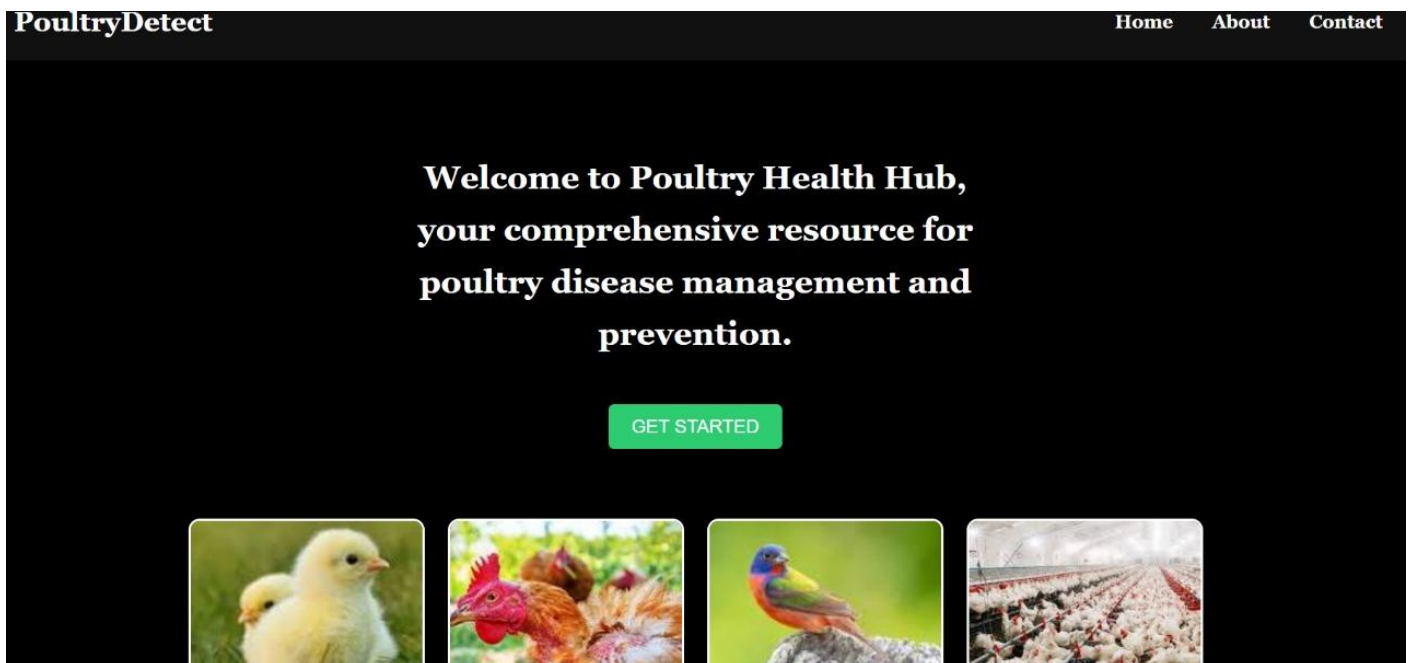
Run the application:

- Open the Anaconda prompt from the start menu.
- Navigate to the folder where your Python script is.
- Now type the "python my_app.py" command.
- Navigate to the localhost where you can view your web page.

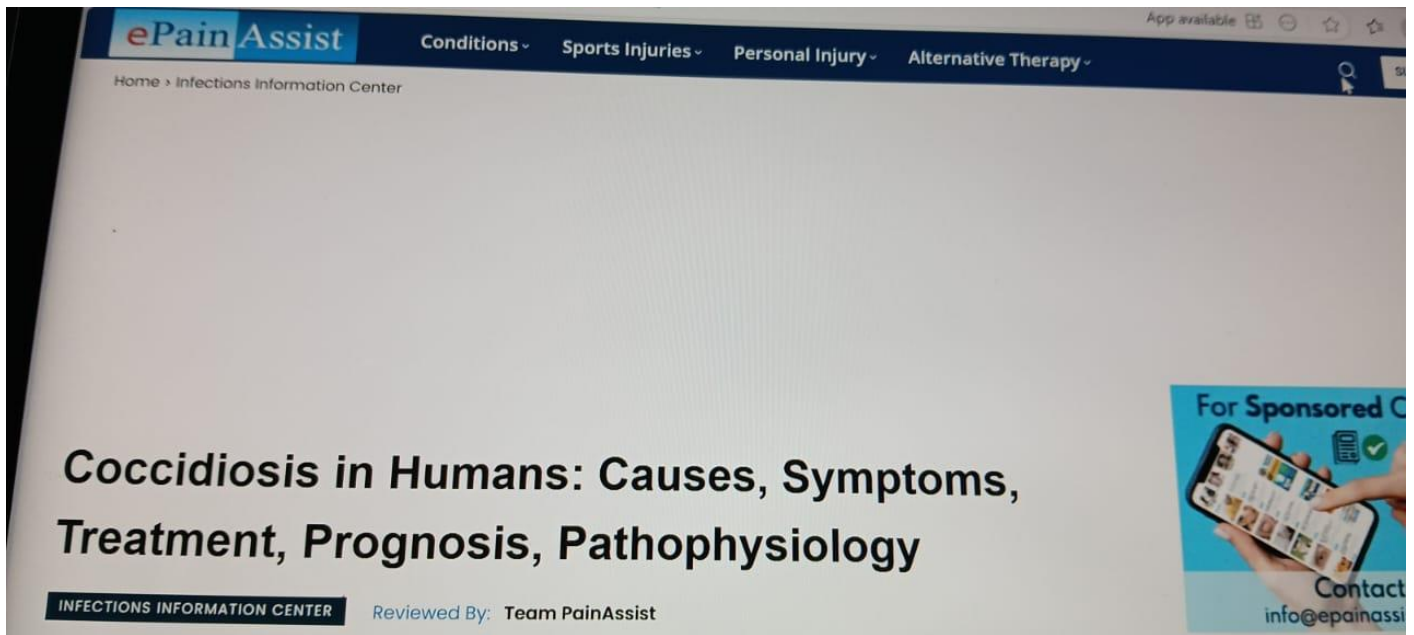
- Click on the predict button from the top right corner, enter the inputs, click on the submit button, and see the result/prediction on the web.

```
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 AVX512F AVX512_VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.  
* Debugger is active!  
* Debugger PIN: 946-817-010  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Click on Get Started at the top right corner to go to the inner page as below:



Output2:



FUTURE ENHANCEMENTS:

✓ We can provide our basic requirements how our design should be, it will directly provide the best top most designated structure for the customer satisfaction.

✓ We can add the AI chatbot to this basic classification fabrics using deep learning

Project which benefits the easy interface for all the new users and the bugs can be easily Solved in time.

✓ We can add the all the textile companies collab with one-platform to interact with Each other and increase their Business strategies.

✓ By providing the all the textile related information in one-platform all the small textile Producers can be benefitted more.

✓ Can be add tie-up with foreign traders for the export and import the textiles which

Is not only benefit to the textile industries but also increase our Indian economy growth.

THE END

vv