

Food Image Classification

Context:

Image classification has become less complicated with deep learning and the availability of larger datasets and computational assets. The Convolution neural network is the most popular and extensively used image classification technique in the latest day.

Clicks is a stock photography company and is an online source of images available for people and companies to download. Photographers from all over the world upload food-related images to the stock photography agency every day. Since the volume of the images that get uploaded daily will be high, it will be difficult for anyone to label the images manually.

Objective:

Clicks have decided to use only three categories of food (**Bread**, **Soup**, and **Vegetables-Fruits**) for now, and you as a data scientist at Clicks, need to build a classification model using a dataset consisting of images that would help to label the images into different categories.

Dataset:

The dataset folder contains different food images. The images are already split into Training and Testing folders. Each folder has three subfolders named **Bread**, **Soup**, and **Vegetables-Fruits**. These folders have images of the respective classes.

Mount the Drive

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

Importing the Libraries

```
In [1]: import numpy as np
import pandas as pd
```

```

import matplotlib.pyplot as plt
import os
import zipfile

# For Data Visualization
import cv2
import seaborn as sns

# For Model Building
import tensorflow as tf
import keras
from tensorflow.keras.models import Sequential, Model # Sequential API for s
from tensorflow.keras.layers import Dense, Dropout, Flatten # Importing dift
from tensorflow.keras.layers import Conv2D, MaxPooling2D, BatchNormalizatio
from tensorflow.keras import backend
from tensorflow.keras.utils import to_categorical # To perform one-hot encod
from tensorflow.keras.optimizers import RMSprop, Adam, SGD # Optimizers for
from tensorflow.keras.callbacks import EarlyStopping # Regularization metho
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras import losses, optimizers
from tensorflow.keras.preprocessing.image import load_img
#from google.colab.patches import cv2_imshow

```

Importing the Dataset

Instructions to access the data through Google Colab:

Follow the below steps:

1. Download the zip file from Olympus.
2. Upload the file into your drive and unzip the folder using the code provided in the notebook. Do not unzip it manually.
3. Please check that the name of the file in your Google Drive is 'Food_Data.zip'. If it's not, then you may rename it on your drive or change it in the following cell.
4. Now, you can run the following cell. If all the earlier steps were done correctly, the dataset will be imported without any errors.

Storing the path of the data file from the Google drive

```
path = 'Food_Data.zip'
```

The data is provided as a zip file so we need to extract the files from the zip file

```
with zipfile.ZipFile(path, 'r') as zip_ref: zip_ref.extractall()
```

Preparing the Data

The dataset has two folders, i.e., 'Training' and 'Testing'. Each of these folders has three sub-folders, namely 'Bread', 'Soup', and 'Vegetable-Fruit'. We will have the Training and Testing path stored in a variable named 'DATADIR'. The names of the sub-folders, which will be the classes for our classification task will be stored in an array called 'CATEGORIES'.

Training Data

We will convert each image into arrays and store them in an array called 'training_data' along with their class index.

```
In [2]: # Storing the training path in a variable named DATADIR, and storing the uni

DATADIR = "Food_Data/Training"                                # Path
CATEGORIES = ["Bread", "Soup", "Vegetable-Fruit"]
IMG_SIZE = 150
```

```
In [3]: # Here we will be using a user defined function create_training_data() to ex
training_data = []

# Storing all the training images
def create_training_data():
    for category in CATEGORIES:
        path = os.path.join(DATADIR, category)
        class_num = category

        for img in os.listdir(path):
            img_array = cv2.imread(os.path.join(path, img))

            new_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))

            training_data.append([new_array, class_num])

create_training_data()
```

Testing Data

We will do the same operation with our Testing data. We will convert each images into arrays and then append them to our array named 'testing_data' along with their class indexes.

```
In [4]: DATADIR_test = "Food_Data/Testing"                    # Path
CATEGORIES = ["Bread", "Soup", "Vegetable-Fruit"]
```

```
IMG_SIZE = 150
```

```
In [5]: # Here we will be using a user defined function create_testing_data() to ext
testing_data = []

# Storing all the testing images
def create_testing_data():
    for category in CATEGORIES:
        path = os.path.join(DATADIR_test, category)
        class_num = category

        for img in os.listdir(path):
            img_array = cv2.imread(os.path.join(path, img))

            new_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))

            testing_data.append([new_array, class_num])

create_testing_data()
```

Visualizing images randomly from each class

Bread

```
In [6]: bread_imgs = [fn for fn in os.listdir(f'{DATADIR}/{CATEGORIES[0]}')]
select_bread = np.random.choice(bread_imgs, 9, replace = False)

fig = plt.figure(figsize = (10, 10))

for i in range(9):
    ax = fig.add_subplot(4, 3, i + 1)

    fp = f'{DATADIR}/{CATEGORIES[0]}/{select_bread[i]}'

    fn = load_img(fp, target_size = (150, 150))

    plt.imshow(fn, cmap = 'Greys_r')

    plt.axis('off')

plt.show()
```



Observations:

- Most bread items have a round, oval or elliptical shape, except for sandwiches.
- Almost all bread items have a grilled or charred portion, which can be an easily recognizable feature to our Neural Network.

Soup

```
In [7]: soup_imgs = [fn for fn in os.listdir(f'{DATADIR}/{CATEGORIES[1]}')]
select_soup = np.random.choice(soup_imgs, 9, replace = False)

fig = plt.figure(figsize = (10, 10))

for i in range(9):
    ax = fig.add_subplot(4, 3, i + 1)

    fp = f'{DATADIR}/{CATEGORIES[1]}/{select_soup[i]}'

    fn = load_img(fp, target_size = (150, 150))
```

```
plt.imshow(fn, cmap = 'Greys_r')

plt.axis('off')

plt.show()
```



Observations:

- All Soup images are defined by a liquid taking on the shape of the container or utensil it is kept in.
- There is a distinct glare from the reflection of light on most of the images.
- Also, almost all of these images have a utensil, which can be a feature that confuses the model between bread and soup. As, images from both the classes mostly contain a dish or a bowl, where they are placed.

Vegetable-Fruit

```
In [8]: vegetable_fruit_imgs = [fn for fn in os.listdir(f'{DATADIR}/{CATEGORIES[2]}')]
select_vegetable_fruit = np.random.choice(vegetable_fruit_imgs, 9, replace =
```



```

fig = plt.figure(figsize = (10, 10))

for i in range(9):
    ax = fig.add_subplot(4, 3, i + 1)

    fp = f'{DATADIR}/{CATEGORIES[2]}/{select_vegetable_fruit[i]}'

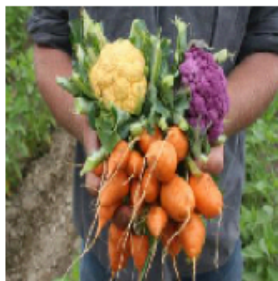
    fn = load_img(fp, target_size = (150, 150))

    plt.imshow(fn, cmap = 'Greys_r')

    plt.axis('off')

plt.show()

```



Observation:

- Most of the images in these classes have vibrant colors and a repeating shape throughout the image.

Data Preprocessing

The arrays `training_data` and `testing_data` had the images stored as arrays with their corresponding labels as the class indexes. So in essence, our `training_data` and

testing_data were arrays of tuples, where each tuple contained the image and its label.

In the following cells, we will unpack the tuples. We will shuffle our training_data and testing_data, and store the images in X_train, and X_test, and the labels in y_train, and y_test respectively.

```
In [9]: # Creating two different lists to store the Numpy arrays and the correspondi
X_train = []
y_train = []

np.random.shuffle(training_data)
for features, label in training_data:
    X_train.append(features)
    y_train.append(label)
```

```
In [10]: # Creating two different lists to store the Numpy arrays and the correspondi
X_test = []
y_test = []

np.random.shuffle(testing_data)
for features, label in testing_data:
    X_test.append(features)
    y_test.append(label)
```

```
In [11]: # Converting the pixel values into Numpy array
X_train = np.array(X_train)
X_test = np.array(X_test)
X_train.shape
```

```
Out[11]: (3203, 150, 150, 3)
```

Note: Images are digitally represented in the form of NumPy arrays which can be observed from the X_train values generated above, so it is possible to perform all the preprocessing operations and build our CNN model using NumPy arrays directly. So, even if the data is provided in the form of NumPy arrays rather than images, we can use this to work on our model.

```
In [12]: # Converting the lists into DataFrames
y_train = pd.DataFrame(y_train, columns = ["Label"], dtype = object)
y_test = pd.DataFrame(y_test, columns = ["Label"], dtype = object)
```

Since the given data is stored in variables X_train, X_test, y_train, and y_test, there is no need to split the data further.

Checking Distribution of Classes

```
In [13]: # Printing the value counts of target variable
count = y_train.Label.value_counts()
print(count)
```



```
print('*'*10)

count = y_train.Label.value_counts(normalize = True)
print(count)
```

```
Label
Soup          1500
Bread          994
Vegetable-Fruit 709
Name: count, dtype: int64
*****
Label
Soup          0.468311
Bread          0.310334
Vegetable-Fruit 0.221355
Name: proportion, dtype: float64
```

Normalizing the data

In neural networks, it is always suggested to **normalize the feature inputs**.

Normalization has the below benefits while training the model of a neural network:

1. **Normalization makes the training faster and reduces the chances of getting stuck at local optima.**
2. In deep neural networks, **normalization helps to avoid exploding gradient problems**. Gradient exploding problem occurs when large error gradients accumulate and result in very large updates to neural network model weights during training. This makes a model unstable and unable to learn from the training data.

As we know image pixel **values range from 0-255**, here we are simply **dividing all the pixel values by 255 to standardize all the images to have values between 0-1**.

```
In [14]: # Normalizing the image data
X_train = X_train/255.0
X_test = X_test/255.0
```

Encoding Target Variable

For any ML or DL techniques, the labels must be encoded into numbers or arrays, so that we can compute the cost between the predicted and the real labels.

In this case, we have 3 classes "Bread", "Soup", and "Vegetable-Fruit". We want the corresponding labels to look like:

- [1, 0, 0] ----- Bread
- [0, 1, 0] ----- Soup
- [0, 0, 1] ----- Vegetable-Fruit

Each class will be represented in the form of an array.

```
In [15]: y_train_encoded = [ ]

for label_name in y_train["Label"]:
    if(label_name == 'Bread'):
        y_train_encoded.append(0)

    if(label_name == 'Soup'):
        y_train_encoded.append(1)

    if(label_name == 'Vegetable-Fruit'):
        y_train_encoded.append(2)

y_train_encoded = to_categorical(y_train_encoded, 3)
y_train_encoded
```

```
Out[15]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 1., 0.],
                ...,
                [1., 0., 0.],
                [1., 0., 0.],
                [0., 1., 0.]])
```

```
In [16]: y_test_encoded = [ ]

for label_name in y_test["Label"]:
    if(label_name == 'Bread'):
        y_test_encoded.append(0)

    if(label_name == 'Soup'):
        y_test_encoded.append(1)

    if(label_name == 'Vegetable-Fruit'):
        y_test_encoded.append(2)

y_test_encoded = to_categorical(y_test_encoded, 3)
y_test_encoded
```

```
Out[16]: array([[0., 1., 0.],
                [0., 1., 0.],
                [0., 1., 0.],
                ...,
                [1., 0., 0.],
                [0., 0., 1.],
                [0., 0., 1.]])
```

Model Building

Now that we have done data preprocessing, let's build the first Convolutional Neural Network (CNN) model.

Model 1 Architecture:

- The first CNN Model will have three convolutional blocks.
- Each convolutional block will have a Conv2D layer and a MaxPooling2D Layer.
- Add first Conv2D layer with **64 filters** and a **kernel size of 3x3**. Use the **'same' padding** and provide the **input shape = (150, 150, 3)**. Use **'relu' activation**.
- Add MaxPooling2D layer with **kernel size 2x2** and use **padding = 'same'**.
- Add a second Conv2D layer with **32 filters** and a **kernel size of 3x3**. Use the **'same' padding** and **'relu' activation**.
- Follow it up with another MaxPooling2D layer **kernel size 2x2** and use **padding = 'same'**.
- Add a third Conv2D layer with **32 filters and the kernel size of 3x3**. Use the **'same' padding** and **'relu' activation**. Once again, follow it up with another Maxpooling2D layer with **kernel size 2x2** and **padding = 'same'**.
- Once the convolutional blocks are added, add the Flatten layer.
- Finally, add dense layers.
- Add first Dense layer with **100 neurons** and **'relu' activation**
- The last dense layer must have as many neurons as the number of classes, which in this case is 3 and use **'softmax' activation**.
- Initialize SGD optimizer with **learning rate = 0.01** and **momentum = 0.9**
- Compile your model using the optimizer you initialized and use **categorical_crossentropy** as the loss function and 'accuracy' as the metric
- Print the model summary and write down your observations/insights about the model.

Note: We need to clear the previous model's history from the Keras backend. Also, we must fix the seed for random number generators after clearing the backend to make sure we receive the same output every time we run the code.

```
In [17]: from tensorflow.keras import backend
backend.clear_session()

# Fixing the seed for random number generators so that we can ensure we receive the same output
np.random.seed(42)
import random
random.seed(42)
tf.random.set_seed(42)
```

```
In [18]: # Initializing a sequential model
model = Sequential()

# Adding first conv layer with 64 filters and kernel size 3x3, padding 'same'
# The input_shape denotes input image dimension
model.add(Conv2D(64, (3, 3), activation = 'relu', padding = "same", input_shape=(150, 150, 3)))

# Adding max pooling to reduce the size of output of first conv layer
```

```

model.add(MaxPooling2D((2, 2), padding = 'same'))

# Adding second conv layer with 32 filters and kernel size 3x3, padding 'same'
model.add(Conv2D(32, (3, 3), activation = 'relu', padding = "same"))
model.add(MaxPooling2D((2, 2), padding = 'same'))

# Add third conv layer with 32 filters and kernel size 3x3, padding 'same'
model.add(Conv2D(32, (3, 3), activation='relu', padding="same"))
model.add(MaxPooling2D((2, 2), padding = 'same'))

# Flattening the output of the conv layer after max pooling to make it ready
model.add(Flatten())

# Adding a fully connected dense layer with 100 neurons
model.add(Dense(100, activation = 'relu'))

# Adding the output layer with 3 neurons and activation functions as softmax
model.add(Dense(3, activation = 'softmax'))

# Using SGD Optimizer
opt = SGD(learning_rate = 0.01, momentum = 0.9)

# Compiling the model
model.compile(optimizer = opt, loss = 'categorical_crossentropy', metrics =

# Generating the summary of the model
model.summary()

```

```

/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/keras/src/
layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_
shape`/`input_dim` argument to a layer. When using Sequential models, prefer
using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2025-02-19 22:46:17.509106: I metal_plugin/src/device/metal_device.cc:1154]
Metal device set to: Apple M3 Max
2025-02-19 22:46:17.509129: I metal_plugin/src/device/metal_device.cc:296] s
ystemMemory: 36.00 GB
2025-02-19 22:46:17.509133: I metal_plugin/src/device/metal_device.cc:313] m
axCacheSize: 13.50 GB
WARNING: All log messages before absl::InitializeLog() is called are written
to STDERR
I0000 00:00:1740026777.509145 18523597 pluggable_device_factory.cc:305] Coul
d not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel
may not have been built with NUMA support.
I0000 00:00:1740026777.509161 18523597 pluggable_device_factory.cc:271] Crea
ted TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 M
B memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id: <
undefined>)

```

Model: "sequential"

Layer (type)	Output Shape	Par
conv2d (Conv2D)	(None, 150, 150, 64)	1
max_pooling2d (MaxPooling2D)	(None, 75, 75, 64)	
conv2d_1 (Conv2D)	(None, 75, 75, 32)	18
max_pooling2d_1 (MaxPooling2D)	(None, 38, 38, 32)	
conv2d_2 (Conv2D)	(None, 38, 38, 32)	9
max_pooling2d_2 (MaxPooling2D)	(None, 19, 19, 32)	
flatten (Flatten)	(None, 11552)	
dense (Dense)	(None, 100)	1,155
dense_1 (Dense)	(None, 3)	

Total params: 1,185,107 (4.52 MB)

Trainable params: 1,185,107 (4.52 MB)

Non-trainable params: 0 (0.00 B)

Observations:

- As we can see from the above summary, this CNN model will train and learn **1,185,107 parameters (weights and biases)**.
- There are **no non-trainable parameters** in the model.
- The model is fairly large and we might expect overfitting.

Training the Model


Let's now train the model using the training data.

```
In [19]: # The following lines of code saves the best model's parameters if training
es = EarlyStopping(monitor = 'val_loss', mode = 'min', verbose = 1, patience
mc = ModelCheckpoint('best_model.h5', monitor = 'val_accuracy', mode = 'max'


# Fitting the model with 30 epochs and validation_split as 10%
history=model.fit(X_train,
                  y_train_encoded,
                  epochs = 60,
                  batch_size= 32, validation_split = 0.10, callbacks = [es, mc])
```


Epoch 1/60


2025-02-19 22:46:18.153942: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enabled.


91/91  0s 20ms/step - accuracy: 0.4336 - loss: 1.0736
Epoch 1: val_accuracy improved from -inf to 0.45171, saving model to best_model.h5


WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.


91/91  3s 23ms/step - accuracy: 0.4341 - loss: 1.0731 - val_accuracy: 0.4517 - val_loss: 1.2266
Epoch 2/60


88/91  0s 19ms/step - accuracy: 0.4368 - loss: 1.1116
Epoch 2: val_accuracy did not improve from 0.45171


91/91  2s 20ms/step - accuracy: 0.4380 - loss: 1.1094 - val_accuracy: 0.4517 - val_loss: 1.0876
Epoch 3/60


88/91  0s 19ms/step - accuracy: 0.4485 - loss: 1.0727
Epoch 3: val_accuracy did not improve from 0.45171


91/91  2s 20ms/step - accuracy: 0.4494 - loss: 1.0716 - val_accuracy: 0.4517 - val_loss: 1.0612
Epoch 4/60

88/91  0s 19ms/step - accuracy: 0.4728 - loss: 1.0383
Epoch 4: val_accuracy did not improve from 0.45171


91/91  2s 20ms/step - accuracy: 0.4740 - loss: 1.0373 - val_accuracy: 0.4517 - val_loss: 1.0625
Epoch 5/60


91/91  0s 19ms/step - accuracy: 0.4540 - loss: 1.0488
Epoch 5: val_accuracy did not improve from 0.45171

91/91  2s 21ms/step - accuracy: 0.4543 - loss: 1.0486 - val_accuracy: 0.4455 - val_loss: 1.0221
Epoch 6/60

88/91  0s 19ms/step - accuracy: 0.5059 - loss: 0.9689
Epoch 6: val_accuracy improved from 0.45171 to 0.59190, saving model to best_model.h5

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

91/91  2s 21ms/step - accuracy: 0.5078 - loss: 0.9662 - val_accuracy: 0.5919 - val_loss: 0.8406
Epoch 7/60

88/91  0s 20ms/step - accuracy: 0.5127 - loss: 0.9690
Epoch 7: val_accuracy improved from 0.59190 to 0.65732, saving model to best_model.h5

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.


```

91/91 _____ 2s 21ms/step - accuracy: 0.5144 - loss: 0.9659 -
val_accuracy: 0.6573 - val_loss: 0.7157
Epoch 8/60
91/91 _____ 0s 20ms/step - accuracy: 0.6137 - loss: 0.7714
Epoch 8: val_accuracy did not improve from 0.65732
91/91 _____ 2s 21ms/step - accuracy: 0.6138 - loss: 0.7714 -
val_accuracy: 0.6293 - val_loss: 0.7276
Epoch 9/60
88/91 _____ 0s 20ms/step - accuracy: 0.6685 - loss: 0.6935
Epoch 9: val_accuracy did not improve from 0.65732
91/91 _____ 2s 21ms/step - accuracy: 0.6688 - loss: 0.6937 -
val_accuracy: 0.6355 - val_loss: 0.7830
Epoch 10/60
88/91 _____ 0s 20ms/step - accuracy: 0.6837 - loss: 0.6776
Epoch 10: val_accuracy improved from 0.65732 to 0.67601, saving model to bes
t_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g. `model.save('my_model.
keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
91/91 _____ 2s 21ms/step - accuracy: 0.6846 - loss: 0.6766 -
val_accuracy: 0.6760 - val_loss: 0.7048
Epoch 11/60
88/91 _____ 0s 20ms/step - accuracy: 0.7246 - loss: 0.6360
Epoch 11: val_accuracy did not improve from 0.67601
91/91 _____ 2s 21ms/step - accuracy: 0.7249 - loss: 0.6356 -
val_accuracy: 0.6667 - val_loss: 0.7924
Epoch 12/60
88/91 _____ 0s 20ms/step - accuracy: 0.7296 - loss: 0.6401
Epoch 12: val_accuracy did not improve from 0.67601
91/91 _____ 2s 21ms/step - accuracy: 0.7285 - loss: 0.6432 -
val_accuracy: 0.6106 - val_loss: 0.8570
Epoch 13/60
88/91 _____ 0s 20ms/step - accuracy: 0.7183 - loss: 0.6742
Epoch 13: val_accuracy did not improve from 0.67601
91/91 _____ 2s 21ms/step - accuracy: 0.7195 - loss: 0.6713 -
val_accuracy: 0.6667 - val_loss: 0.7975
Epoch 14/60
91/91 _____ 0s 21ms/step - accuracy: 0.7715 - loss: 0.5791
Epoch 14: val_accuracy did not improve from 0.67601
91/91 _____ 2s 22ms/step - accuracy: 0.7714 - loss: 0.5796 -
val_accuracy: 0.6636 - val_loss: 1.0905
Epoch 15/60
89/91 _____ 0s 21ms/step - accuracy: 0.7345 - loss: 0.7859
Epoch 15: val_accuracy did not improve from 0.67601
91/91 _____ 2s 22ms/step - accuracy: 0.7345 - loss: 0.7860 -
val_accuracy: 0.6667 - val_loss: 1.2958
Epoch 15: early stopping

```

The following lines of code saves the best model's parameters if training accuracy goes down on further training

```
es = EarlyStopping(monitor = 'val_loss', mode = 'min', verbose = 1, patience = 5) mc =  
ModelCheckpoint('best_model.h5', monitor = 'val_accuracy', mode = 'max', verbose = 1,  
save_best_only = True)
```

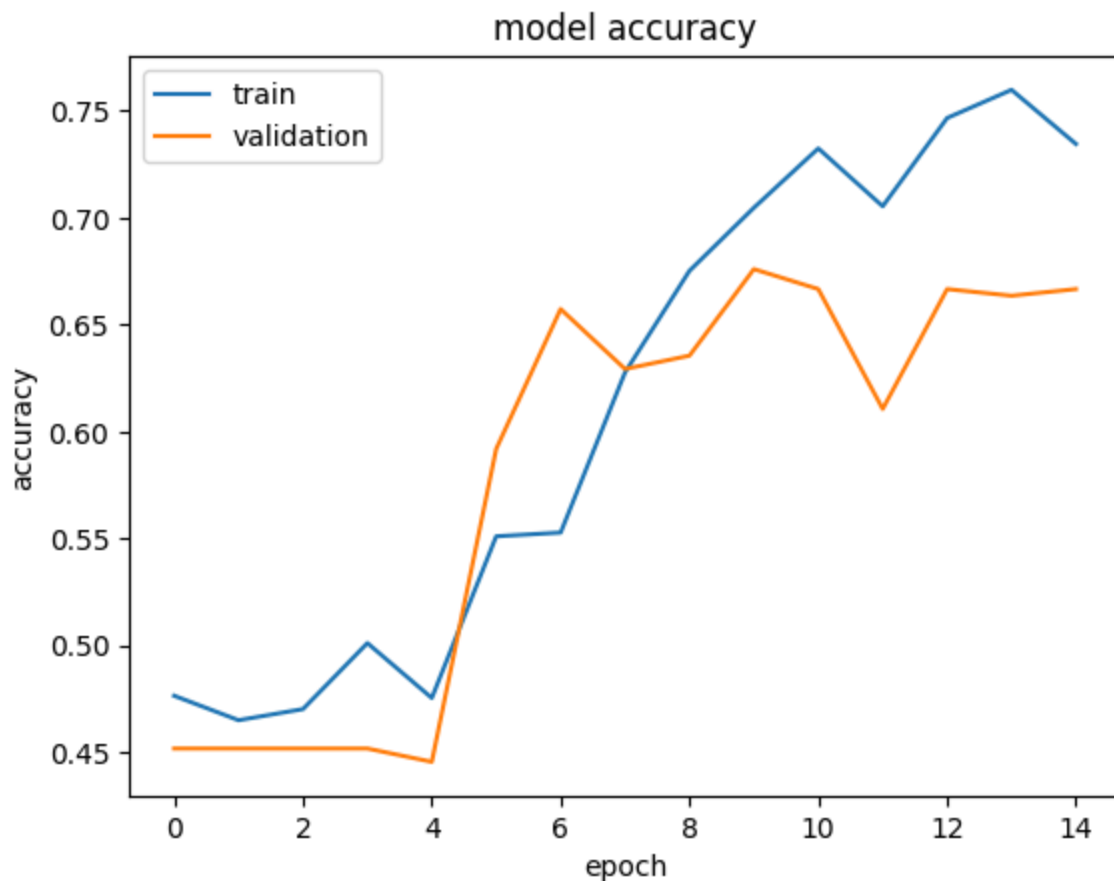
Fitting the model with 30 epochs and validation_split as 10%

```
history=model.fit(X_train, y_train_encoded, epochs = 60, batch_size= 32,  
validation_split = 0.1, callbacks = [es, mc])
```

Plotting the Training and Validation Accuracies

```
In [20]: # Plotting the training and validation accuracies for each epoch
```

```
plt.plot(history.history['accuracy'])  
plt.plot(history.history['val_accuracy'])  
plt.title('model accuracy')  
plt.ylabel('accuracy')  
plt.xlabel('epoch')  
plt.legend(['train', 'validation'], loc='upper left')  
plt.show()
```



Checking Test Accuracy

```
In [21]: model.evaluate(X_test, (y_test_encoded))
```

35/35 ————— 0s 7ms/step – accuracy: 0.6494 – loss: 1.4614

```
Out[21]: [1.408872365951538, 0.64716637134552]
```

Observations:

- The training didn't continue for all of the 60 epochs. The training stopped because the performance wasn't improving beyond a certain point.
- From the above plot, we observe that the training accuracy is continuously improving. However, it was not the case with the validation accuracy. The validation accuracy started fluctuating after 5 epochs.
- All the above observations suggest that the model was overfitting on the training data.
- However, the model was consistent on validation and test data.

Plotting Confusion Matrix

```
In [22]: from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

pred = model.predict(X_test)
pred = np.argmax(pred, axis = 1)
y_true = np.argmax(y_test_encoded, axis = 1)

# Printing the classification report
print(classification_report(y_true, pred))

# Plotting the heatmap using confusion matrix
cm = confusion_matrix(y_true, pred)
plt.figure(figsize = (8, 5))
sns.heatmap(cm, annot = True, fmt = '.0f', xticklabels = ['Bread', 'Soup',
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

35/35 ————— 0s 5ms/step

	precision	recall	f1-score	support
0	0.78	0.14	0.24	362
1	0.66	0.90	0.76	500
2	0.60	0.88	0.72	232
accuracy			0.65	1094
macro avg	0.68	0.64	0.57	1094
weighted avg	0.68	0.65	0.58	1094



Observations:

- The model is giving about 70% accuracy on the test data
- There have been many misclassifications between all classes.
- A large number of images of 'Bread' were predicted to be 'Soup'. We had earlier predicted this because both these classes show the presence of a dish or a utensil in the images.
- There have been misclassifications between 'Bread' and 'Vegetable-Fruit' as well. We can attribute this to the presence of yellowish pixels in both. Hence, the model might have taken one for the other.
- The misclassifications between 'Vegetable-Fruit' and 'Soup' have been the least, as we can see that there is minimal visual overlap among these classes.

Let's try to build another model with a different architecture and see if we can improve the model performance. Since the first model was overfitting, we will add Dropout layers at the end of each convolutional block.

Model 2 Architecture:

- We plan on having 4 convolutional blocks in this Architecture, each having a Conv2D, MaxPooling2D, and a Dropout layer.
- Add first Conv2D layer with **256 filters** and a **kernel size of 5x5**. Use the **'same' padding** and provide the **input shape = (150, 150, 3)**. Use **'relu' activation**.

- Add MaxPooling2D layer with **kernel size 2x2** and **stride size 2x2**.
- Add a Dropout layer with a dropout ratio of **0.25**.
- Add a second Conv2D layer with **128 filters** and a **kernel size of 5x5**. Use the **'same' padding** and **'relu' activation**.
- Follow this up with a similar Maxpooling2D layer like above and a Dropout layer with 0.25 dropout ratio.
- Add a third Conv2D layer with **64 filters** and a **kernel size of 3x3**. Use the **'same' padding** and **'relu' activation**.
- Follow this up with a similar Maxpooling2D layer and a Dropout layer with dropout ratio of 0.25.
- Add a fourth Conv2D layer with **32 filters** and a **kernel size of 3x3**. Use the **'same' padding** and **'relu' activation**.
- Follow this up with a similar Maxpooling2D layer and a Dropout layer with dropout ratio of 0.25.
- Once the convolutional blocks are added, add the Flatten layer.
- Add first fully connected dense layer with 64 neurons and use **'relu' activation**.
- Add a second fully connected dense layer with 32 neurons and use **'relu' activation**.
- Add your final dense layer with 3 neurons and use **'softmax' activation function**.
- Initialize an **Adam optimizer** with a learning rate of 0.001.
- Compile your model with the optimizer you initialized and use **categorical_crossentropy** as the loss function and the 'accuracy' as the metric.
- Print your model summary and write down your observations.

```
In [23]: from tensorflow.keras import backend
backend.clear_session()

# Fixing the seed for random number generators so that we can ensure we receive the same results
np.random.seed(42)
import random
random.seed(42)
tf.random.set_seed(42)
```

```
In [24]: # Initializing a sequential model
model_2 = Sequential()

# Adding first conv layer with 256 filters and kernel size 5x5, with ReLU activation
# The input_shape denotes input image dimension
model_2.add(Conv2D(filters = 256, kernel_size = (5, 5), padding = 'Same', activation='relu', input_shape=(28, 28, 1)))

# Adding max pooling to reduce the size of output of first conv layer
model_2.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))

# Adding dropout to randomly switch off 25% neurons to reduce overfitting
model_2.add(Dropout(0.25))

# Adding second conv layer with 128 filters and with kernel size 5x5 and ReLU activation
model_2.add(Conv2D(filters = 128, kernel_size = (5, 5), padding = 'Same', activation='relu'))
```

```

# Adding max pooling to reduce the size of output of first conv layer
model_2.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))

# Adding dropout to randomly switch off 25% neurons to reduce overfitting
model_2.add(Dropout(0.25))

# Adding third conv layer with 64 filters and with kernel size 3x3 and ReLu
model_2.add(Conv2D(filters = 64, kernel_size = (3, 3), padding = 'Same', act

# Adding max pooling to reduce the size of output of first conv layer
model_2.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))

# Adding dropout to randomly switch off 25% neurons to reduce overfitting
model_2.add(Dropout(0.25))

# Adding fourth conv layer with 32 filters and with kernel size 3x3 and ReLu
model_2.add(Conv2D(filters = 32, kernel_size = (3, 3), padding = 'Same', act

# Adding max pooling to reduce the size of output of first conv layer
model_2.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))

# Adding dropout to randomly switch off 25% neurons to reduce overfitting
model_2.add(Dropout(0.25))

# Flattening the 3-d output of the conv layer after max pooling to make it r
model_2.add(Flatten())

# Adding first fully connected dense layer with 64 neurons
model_2.add(Dense(64, activation = "relu"))

# Adding second fully connected dense layer with 32 neurons
model_2.add(Dense(32, activation = "relu"))

# Adding the output layer with 3 neurons and activation functions as softmax
model_2.add(Dense(3, activation = "softmax"))

# Using Adam Optimizer
optimizer = Adam(learning_rate = 0.001)

# Compile the model
model_2.compile(optimizer = optimizer , loss = "categorical_crossentropy", m

```

```

/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/keras/src/
layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_
shape`/`input_dim` argument to a layer. When using Sequential models, prefer
using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

```

In [25]: `model_2.summary()`

Model: "sequential"

Layer (type)	Output Shape	Par
conv2d (Conv2D)	(None, 150, 150, 256)	19
max_pooling2d (MaxPooling2D)	(None, 75, 75, 256)	
dropout (Dropout)	(None, 75, 75, 256)	
conv2d_1 (Conv2D)	(None, 75, 75, 128)	819
max_pooling2d_1 (MaxPooling2D)	(None, 37, 37, 128)	
dropout_1 (Dropout)	(None, 37, 37, 128)	
conv2d_2 (Conv2D)	(None, 37, 37, 64)	73
max_pooling2d_2 (MaxPooling2D)	(None, 18, 18, 64)	
dropout_2 (Dropout)	(None, 18, 18, 64)	
conv2d_3 (Conv2D)	(None, 18, 18, 32)	18
max_pooling2d_3 (MaxPooling2D)	(None, 9, 9, 32)	
dropout_3 (Dropout)	(None, 9, 9, 32)	
flatten (Flatten)	(None, 2592)	
dense (Dense)	(None, 64)	165
dense_1 (Dense)	(None, 32)	2
dense_2 (Dense)	(None, 3)	

Total params: 1,099,171 (4.19 MB)

Trainable params: 1,099,171 (4.19 MB)

Non-trainable params: 0 (0.00 B)

Observations:

- We can observe from the above summary that this CNN model will train and learn 1,099,171 parameters (weights and biases).** However, since we have Dropout layers, as the training progresses, few of the neurons will be dropped and thus effective trainable parameters will also be less.
- This model has more convolutional blocks and hence, we can expect this model to perform better in extracting features from the images.
- We are using a different optimizer. i.e., Adam. Let's see if we receive any improvement in performance.

Training the Model

Let's now train the model using the training data.

```
In [26]: es = EarlyStopping(monitor = 'val_loss', mode = 'min', verbose = 1, patience
mc = ModelCheckpoint('best_model.h5', monitor = 'val_accuracy', mode = 'max'

...
history=model_2.fit(X_train,
                    y_train_encoded,
                    epochs = 60,
                    batch_size = 32, validation_split = 0.10, use_multiprocessing = Tr
...



















history = model_2.fit(
    X_train,
    y_train_encoded,
    epochs=60,
    batch_size=32,
    validation_split=0.10,
    callbacks=[es, mc], # Include callbacks if you want early stopping and
    verbose=1
)
```

Epoch 1/60

91/91  0s 267ms/step - accuracy: 0.4375 - loss: 1.1786

Epoch 1: val_accuracy improved from -inf to 0.45171, saving model to best_model.h5

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.save_model(model, 'my_model.keras')`.

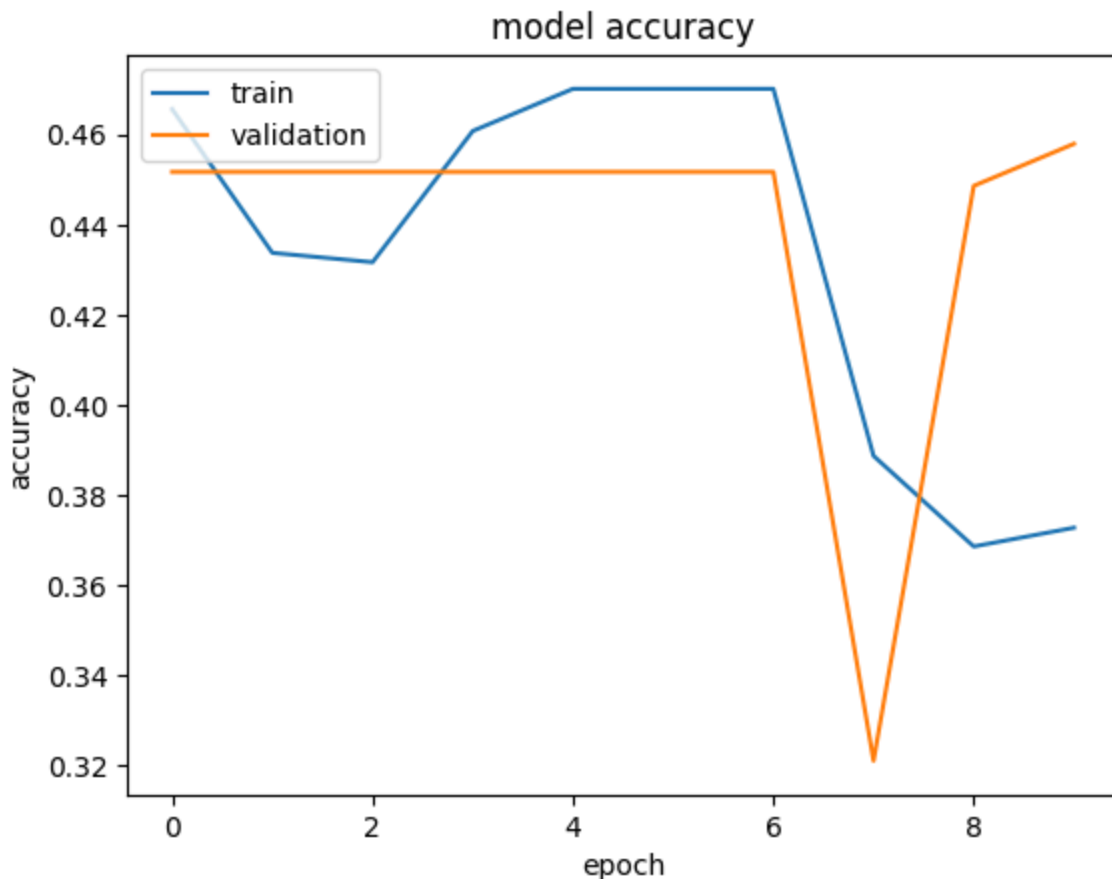
91/91  **26s** 278ms/step - accuracy: 0.4378 - loss: 1.1779
 - val_accuracy: 0.4517 - val_loss: 1.0817
 Epoch 2/60
90/91  **0s** 317ms/step - accuracy: 0.4364 - loss: 1.5066
 Epoch 2: val_accuracy did not improve from 0.45171
91/91  **30s** 325ms/step - accuracy: 0.4363 - loss: 1.5081
 - val_accuracy: 0.4517 - val_loss: 1.0770
 Epoch 3/60
90/91  **0s** 413ms/step - accuracy: 0.4026 - loss: 3.4437
 Epoch 3: val_accuracy did not improve from 0.45171
91/91  **39s** 425ms/step - accuracy: 0.4032 - loss: 3.4196
 - val_accuracy: 0.4517 - val_loss: 1.0690
 Epoch 4/60
91/91  **0s** 611ms/step - accuracy: 0.4432 - loss: 1.0737
 Epoch 4: val_accuracy did not improve from 0.45171
91/91  **57s** 633ms/step - accuracy: 0.4434 - loss: 1.0736
 - val_accuracy: 0.4517 - val_loss: 1.0712
 Epoch 5/60
90/91  **0s** 584ms/step - accuracy: 0.4496 - loss: 1.0699
 Epoch 5: val_accuracy did not improve from 0.45171
91/91  **54s** 592ms/step - accuracy: 0.4500 - loss: 1.0695
 - val_accuracy: 0.4517 - val_loss: 1.0687
 Epoch 6/60
90/91  **0s** 415ms/step - accuracy: 0.4496 - loss: 1.0693
 Epoch 6: val_accuracy did not improve from 0.45171
91/91  **39s** 423ms/step - accuracy: 0.4500 - loss: 1.0689
 - val_accuracy: 0.4517 - val_loss: 1.0692
 Epoch 7/60
90/91  **0s** 364ms/step - accuracy: 0.4496 - loss: 1.0680
 Epoch 7: val_accuracy did not improve from 0.45171
91/91  **34s** 371ms/step - accuracy: 0.4500 - loss: 1.0676
 - val_accuracy: 0.4517 - val_loss: 1.0790
 Epoch 8/60
90/91  **0s** 335ms/step - accuracy: 0.3905 - loss: 2.8848
 Epoch 8: val_accuracy did not improve from 0.45171
91/91  **31s** 342ms/step - accuracy: 0.3905 - loss: 2.8948
 - val_accuracy: 0.3209 - val_loss: 4.1396
 Epoch 9/60
90/91  **0s** 316ms/step - accuracy: 0.3675 - loss: 18.0331
 Epoch 9: val_accuracy did not improve from 0.45171
91/91  **29s** 322ms/step - accuracy: 0.3676 - loss: 18.1017
 - val_accuracy: 0.4486 - val_loss: 24.9983
 Epoch 10/60
90/91  **0s** 302ms/step - accuracy: 0.3655 - loss: 35.5653
 Epoch 10: val_accuracy improved from 0.45171 to 0.45794, saving model to best_model.h5

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
 `keras.saving.save_model(model)`. This file format is considered legacy. We
 recommend using instead the native Keras format, e.g. `model.save('my_model.
 keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

91/91  **28s** 308ms/step - accuracy: 0.3657 - loss: 35.6533
 - val_accuracy: 0.4579 - val_loss: 34.7343
 Epoch 10: early stopping

Plotting the Training and Validation Accuracies

```
In [27]: plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc = 'upper left')
plt.show()
```



Checking Test Accuracy

```
In [28]: model_2.evaluate(X_test, y_test_encoded)
```

35/35 ————— 3s 78ms/step – accuracy: 0.4715 – loss: 35.4397

```
Out[28]: [37.06787872314453, 0.44332724809646606]
```

Observations:

- By comparing the train and validation accuracy, it seems the model is not overfitting as much. So adding Dropout layers definitely proved beneficial.
- The training also ran for more epochs. So training accuracy never stayed stagnant. It showed improvement throughout.

- The validation accuracy stopped showing any significant improvements after about 10 epochs, however, the test accuracy has improved significantly.

Plotting Confusion Matrix

```
In [29]: from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

pred = model_2.predict(X_test)
pred = np.argmax(pred, axis = 1)
y_true = np.argmax(y_test_encoded, axis = 1)

#Printing the classification report
print(classification_report(y_true, pred))

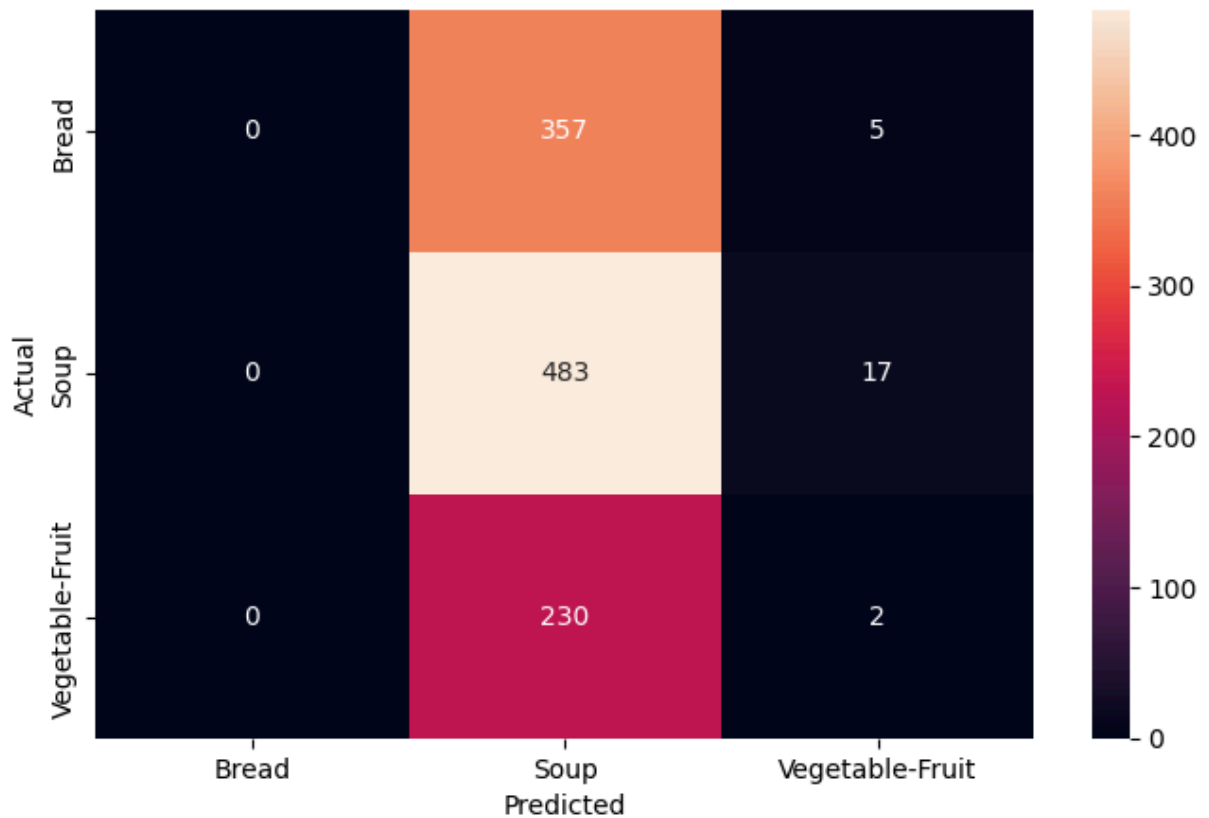
#Plotting the heatmap using confusion matrix
cm = confusion_matrix(y_true, pred)
plt.figure(figsize = (8, 5))
sns.heatmap(cm, annot = True, fmt = '.0f', xticklabels = ['Bread', 'Soup',
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

```
35/35 ————— 3s 71ms/step
              precision    recall  f1-score   support

     0           0.00       0.00       0.00       362
     1           0.45       0.97       0.62       500
     2           0.08       0.01       0.02       232

 accuracy                   0.44       1094
 macro avg           0.18       0.32       0.21       1094
 weighted avg        0.22       0.44       0.28       1094
```

```
/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/sklearn/m
etrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-def
ined and being set to 0.0 in labels with no predicted samples. Use `zero_div
ision` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/sklearn/m
etrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-def
ined and being set to 0.0 in labels with no predicted samples. Use `zero_div
ision` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/sklearn/m
etrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-def
ined and being set to 0.0 in labels with no predicted samples. Use `zero_div
ision` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```



Observations:

- The misclassifications are very less in comparison to the previous model.
- Bread and Soup continue to be the most misclassified classes. However, it's not as bad as the previous model.
- We can still try to add more layers to see if we can bring down the misclassification further.

Prediction

Let us predict using the best model, i.e., model 2, by plotting one random image from `X_test` data and see if our best model is predicting the image correctly or not.

```
In [30]: import matplotlib.pyplot as plt

# Plotting the test image using matplotlib
plt.imshow((X_test[1] * 255).astype('uint8')) # Convert to uint8 for correct display
i = np.argmax(y_test.Label[1])

# Set title based on class index
titles = ["Bread", "Soup", "Vegetable-Fruit"]
plt.title(titles[i])

plt.axis('off')
plt.show()
```


Bread



Convert the image to uint8 and display using OpenCV

```
image = (X_test[1] * 255).astype('uint8') cv2.imshow('Test Image', image)
cv2.waitKey(0) cv2.destroyAllWindows()
```

```
In [31]: # Plotting the test image
#cv2.imshow(X_test[1] * 255) # Multiplying with 255, because X_test was pre
image = (X_test[1] * 255).astype('uint8')
cv2.imshow('Test Image', image)
i = y_test.Label[1]
i = np.argmax(i)
if(i == 0):
    plt.title("Bread")
if(i == 1):
    plt.title("Soup")
if(i == 2):
    plt.title("Vegetable-Fruit")

plt.axis('off')
plt.show()
```

Bread

```
In [32]: # Predicting the test image with the best model and storing the prediction v
res = model_2.predict(X_test[1].reshape(1, 150, 150, 3))
```

1/1 ————— 0s 28ms/step

```
In [33]: # Applying argmax on the prediction to get the highest index value
i=np.argmax(res)
if(i == 0):
    print("Bread")
if(i==1):
    print("Soup")
if(i==2):
    print("Vegetable-Fruit")
```

Soup

Observation:

- The model is able to correct classify the image we have randomly chosen from the test data.

Conclusion and Recommendations

1. As we have seen, the second CNN model was able to predict the test image correctly with a test accuracy of close to 80%.
2. **There is still scope for improvement in the test accuracy of the CNN model** chosen here. **Different architectures** and **optimizers** can be used to build a better

food classifier.

3. Transfer learning can be applied to the dataset to improve accuracy. You can choose among multiple pre-trained models available in the Keras framework.
4. Once the desired performance is achieved from the model, the company can use it to classify different images being uploaded to the website.
5. We can further try to improve the performance of the CNN model by using some of the below techniques and see if you can increase accuracy:
 - We can try hyperparameter tuning for some of the hyperparameters like the number of convolutional blocks, the number of filters in each Conv2D layer, filter size, activation function, adding/removing dropout layers, changing the dropout ratio, etc.
 - Data Augmentation might help to make the model more robust and invariant toward different orientations.