Convolutional Neural Network for Insect Image Classification

November 30, 2023

1 Convolutional Neural Network for Insect Image Classification

1.1 Author

Joel Ruetas

1.2 Abstract

Insects play diverse roles in our environment, with some being harmful by damaging crops, spreading diseases, or causing poisoning, while others, like bees, are crucial for pollination and ecosystem balance. Accurately classifying insects is vital to understanding their impact and managing their effects on human activities and natural ecosystems. This paper explores the use of Convolutional Neural Networks (CNNs) for the classification of insect images. Specifically, we aim to determine whether a CNN can effectively identify whether an insect is harmful or beneficial based solely on its image.

1.3 Objective

The primary objective of this research is to develop and validate a Convolutional Neural Network model capable of accurately classifying images of insects into specific categories, such as harmful or beneficial, based on their visual characteristics. This involves:

- Analyzing and processing a large dataset of insect images from the Natural History Museum, London.
- Utilizing the power of CNNs to learn and recognize patterns and features specific to different insect species.
- Evaluating the model's accuracy and effectiveness in classifying insects, which could have significant implications for environmental management, agriculture, and biological research.

Ultimately, this study aims to contribute to the field of automated image classification in entomology, providing a tool that can assist in the rapid and accurate identification of insect species, thereby enhancing our understanding and management of these crucial components of our ecosystem.

1.4 Dataset

The dataset used in this study comprises high-resolution images of insect specimens from the British carabids collection at the Natural History Museum, London. It includes a vast array of 63,364 specimens across 291 species. Each species is organized into a specific folder, labeled with

the GBIF (Global Biodiversity Information Facility) number corresponding to that species. For instance, the species 'Carabus problematicus' is stored in the folder labeled '4470555'.

Dataset Source: Insect Identification from Habitus Images on Kaggle - **Number of Species**: 291 - **Total Specimens**: 63,364

The GBIF is a global initiative funded by governments worldwide. It provides open access to data about all life forms on Earth, thereby supporting a wide range of biological and environmental research.

1.4.1 Column Descriptions for Dataset

The dataset is structured into the following columns:

- **insect_gbif**: The GBIF identification number for the insect species.
- **path_img**: The file path where the insect image is stored.
- **file_name**: The name of the image file.

1.5 Specimen Properties for Insect Image Classification

1.5.1 Image Characteristics

In this study, the dataset comprises images of insect specimens, each with specific properties that are consistent across the collection. Understanding these properties is crucial for the preprocessing and analysis phases of the Convolutional Neural Network (CNN) model development. The properties of these specimen images include:

- **Format**: All images are in JPEG (jpg) format. JPEG is a commonly used method of lossy compression for digital images, particularly for those images produced by digital photography.
- **Aspect Ratio**: The images maintain an aspect ratio of 640 x 480. This means that the width of the images is 640 pixels, and the height is 480 pixels. The aspect ratio is essential for maintaining consistency in image presentation and is a standard size for medium-resolution images.
- **Size**: Each image is approximately 50 kilobytes (kb) in size. The relatively small size of the images suggests that there might be some level of compression, which can affect the image quality.

1.5.2 Possible Image Issues (Added Noise)

Along with the standard properties, there are certain issues or anomalies present in many of the images, which could add noise or unwanted variability to the dataset:

- **Graininess**: Most of the images exhibit a certain degree of graininess. This refers to the visual appearance of the texture in the images that can resemble a fine granular surface. Graininess can impact the model's ability to discern fine details in the images.
- **Presence of Pins**: Some images include pins, which are likely used for positioning or securing the specimens. These pins can be mistaken as part of the insect by the model if not accounted for during the preprocessing phase.

- **Labels in Images**: Several images contain labels, which might include text or other identifying marks. These labels are extraneous to the insect specimens themselves and can be misleading for the model during the classification process.
- **Specimen Damage**: Some of the specimens are damaged, which means they might not represent the typical appearance of the species. This damage can lead to inaccuracies in classification if the model learns to associate these damages with certain species characteristics.

Given these properties and potential issues, careful preprocessing and augmentation of the images might be necessary. This could involve techniques like noise reduction for graininess, object detection to identify and ignore pins and labels, and careful consideration of how to handle damaged specimens. Addressing these challenges is critical to ensure the CNN model is trained on relevant features and can accurately classify the insect images.

Specimen with pin and label

Specimen with label but without pin

Damaged specimen

1.6 Import Libraries

[ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline

import numpy as np
import os
import pandas as pd
import pickle
import PIL
import random
import seaborn as sns
import tensorflow as tf
import tensorflow_datasets as tfds
import time

from PIL import Image
from sklearn.utils import shuffle
from tqdm.notebook import tqdm

from google.colab import drive
```

```
from keras import layers
from keras.models import Sequential
from keras import backend as K
from keras import metrics
from keras.callbacks import Callback
from tensorflow import keras

tfds.disable_progress_bar()
```

[ ]:
```
# Print the version number of TensorFlow.
print(tf.__version__)
```

```
2.12.0
```

1.7 Mount Google Drive

[ ]:
```
# This command mounts your Google Drive to the Colab Notebook.
# It will ask for your authorization to access the drive.
# Once authorized, you can access your drive files using the path '/content/
  ↪drive'.
drive.mount('/content/drive', force_remount = True)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

1.8 Read the data

Add a method to load images from a folder

[ ]:
```
# Load the paths to the images in a directory
def load_images_from_folder(folder, only_path = False, label = ""):
    if only_path == False:
        images = []
        file_name = []
        for filename in os.listdir(folder):
            img = plt.imread(os.path.join(folder, filename))

            if img is not None:
                end = filename.find(".")
                file_name.append(open[0:end])
                images.append(img)

        return images, file_name
    else:
        path = []
        for filename in os.listdir(folder):
            img_path = os.path.join(folder, filename)
            if img_path is not None:
```

```
                path.append([label, img_path])
        return path
```

1.9 Set up the data and create the dataset

[ ]:
```
# Load the paths on the images
images = []

# Define the path to the directory containing the insect images
path = '/content/drive/MyDrive/Insects/'

# Iterate over each file or directory in the specified path
for f in os.listdir(path):
    # Check if the first file in the directory is a jpg image
    if 'jpg' in os.listdir(path + f)[0]:
        # If it is, load the image paths from this folder using the␣
 ↪'load_images_from_folder' function
        # 'only_path = True' indicates that only the path of the images is␣
 ↪needed, not the images themselves
        # The label for the images is the folder name (f)
        images += load_images_from_folder(folder = path + f, only_path = True,␣
 ↪label = f)

    else:
        # If the first file is not a jpg, it's a subdirectory
        # Iterate over the files in this subdirectory
        for d in os.listdir(path + f):
            # Load the image paths from each subdirectory
            images += load_images_from_folder(folder = path + f + '/' + d,␣
 ↪only_path = True, label = f)


# Create a dataframe with the paths and the label for each insect
df = pd.DataFrame(images, columns = ['insect_gbif', 'path_img']) # Creating a␣
 ↪DataFrame with two columns

# Extract the file name from the image paths and store it in a list
file_name = []
for i in range(len(df['path_img'])):
    # Split the path and extract the file name, then split at '.' and take the␣
 ↪first part (the name without extension)
    temp = df.path_img[i].split('/')[-1].split('.')[0]
    file_name.append(temp)
file_name

# Add the file names as a new column in the DataFrame
```

```
df['file_name'] = file_name
```

[ ]:
```
# Display statistics
df.describe()
```

[ ]:
```
       insect_gbif                                              path_img  \
count        63364                                                 63364
unique         291                                                 63364
top        9364935  /content/drive/MyDrive/Insects/1035167/d162s00…
freq           888                                                     1

        file_name
count       63364
unique      63364
top     d162s0012
freq            1
```

[ ]:
```
# Display summary
df.info()
```

[ ]:
```
df
```

[ ]:
```
      insect_gbif                                              path_img  \
0         1035167  /content/drive/MyDrive/Insects/1035167/d162s00…
1         1035167  /content/drive/MyDrive/Insects/1035167/d162s00…
2         1035167  /content/drive/MyDrive/Insects/1035167/d162s00…
3         1035167  /content/drive/MyDrive/Insects/1035167/d162s00…
4         1035167  /content/drive/MyDrive/Insects/1035167/d162s00…
…             …                                                  …
63359     9581584  /content/drive/MyDrive/Insects/9581584/d135s03…
63360     9581584  /content/drive/MyDrive/Insects/9581584/d135s02…
63361     9581584  /content/drive/MyDrive/Insects/9581584/d135s03…
63362     9581584  /content/drive/MyDrive/Insects/9581584/d135s02…
63363     9581584  /content/drive/MyDrive/Insects/9581584/d135s03…

       file_name
0      d162s0012
1      d162s0016
2      d162s0011
3      d162s0015
4      d162s0013
…          …
63359  d135s0342
63360  d135s0230
63361  d135s0308
63362  d135s0264
63363  d135s0358
```

```
[63364 rows x 3 columns]
```

[ ]:
```
labels = df["insect_gbif"].unique()

labels_dict = dict(zip(range(len(labels)), labels))
labels_dict
```

[ ]:
```
{0: '1035167',
 1: '1035185',
 2: '1035195',
 3: '1035231',
 4: '1035366',
 5: '1035290',
 6: '1035204',
 7: '1035434',
 8: '1035194',
 9: '1035208',
 10: '1035551',
 11: '1035542',
 12: '1035864',
 13: '1035578',
 14: '1035929',
 15: '1035931',
 16: '1036286',
 17: '1036203',
 18: '1036128',
 19: '1036154',
 20: '1036216',
 21: '1036192',
 22: '1036066',
 23: '1036255',
 24: '1036789',
 25: '1036796',
 26: '1037633',
 27: '1037319',
 28: '1036893',
 29: '1037293',
 30: '1036899',
 31: '4308786',
 32: '1036917',
 33: '1036958',
 34: '4308789',
 35: '4308787',
 36: '4308790',
 37: '4308800',
 38: '4308801',
```

```
39: '4308804',
40: '4308805',
41: '4470555',
42: '4308807',
43: '4308812',
44: '4308811',
45: '4308806',
46: '4308815',
47: '4470539',
48: '4470801',
49: '4470765',
50: '4471238',
51: '4471071',
52: '4472849',
53: '4471235',
54: '4472828',
55: '4471269',
56: '4471202',
57: '4471113',
58: '4472858',
59: '4472884',
60: '4473277',
61: '4472929',
62: '4472907',
63: '4473127',
64: '4472897',
65: '4473297',
66: '4472900',
67: '4472913',
68: '4473320',
69: '4473319',
70: '4473974',
71: '4474146',
72: '4473874',
73: '4473795',
74: '4473617',
75: '4473860',
76: '4473834',
77: '4473903',
78: '4474169',
79: '4474653',
80: '4474974',
81: '4474780',
82: '4475140',
83: '4475128',
84: '4474998',
85: '4475099',
```

```
86: '4475127',
87: '4474861',
88: '4475157',
89: '4475179',
90: '4475757',
91: '4475677',
92: '4475188',
93: '4475761',
94: '4475181',
95: '4475213',
96: '4475707',
97: '4475685',
98: '4475794',
99: '4475846',
100: '4475908',
101: '4475902',
102: '4476832',
103: '4476667',
104: '4478699',
105: '4478842',
106: '4479870',
107: '4479052',
108: '4478679',
109: '4478695',
110: '4479890',
111: '4478057',
112: '4479895',
113: '4480200',
114: '4480271',
115: '4480348',
116: '4480315',
117: '4480480',
118: '4988207',
119: '4480502',
120: '4480485',
121: '4480302',
122: '4988363',
123: '4988354',
124: '5716409',
125: '5716408',
126: '4988516',
127: '4988447',
128: '4988484',
129: '5023058',
130: '5023047',
131: '5716406',
132: '5753612',
```

```
133: '5716411',
134: '5753725',
135: '5753697',
136: '5753757',
137: '5753720',
138: '5753653',
139: '5755027',
140: '5755011',
141: '5754974',
142: '5755051',
143: '5755044',
144: '5755066',
145: '5755060',
146: '5755075',
147: '5755079',
148: '5755080',
149: '5755082',
150: '5755302',
151: '5755296',
152: '5755175',
153: '5755451',
154: '5755203',
155: '5755293',
156: '5755339',
157: '5755322',
158: '5755456',
159: '5755562',
160: '5755969',
161: '5756010',
162: '5755986',
163: '5755952',
164: '5755954',
165: '5755950',
166: '5755978',
167: '5755982',
168: '5756015',
169: '5756018',
170: '5756368',
171: '5756021',
172: '5757205',
173: '5756814',
174: '5757155',
175: '5757192',
176: '5757120',
177: '5756945',
178: '5756374',
179: '5756874',
```

```
180: '5757207',
181: '5757211',
182: '5757308',
183: '5757304',
184: '5872109',
185: '5757310',
186: '5872127',
187: '5872124',
188: '5872143',
189: '5872129',
190: '5872148',
191: '5872119',
192: '5872111',
193: '5872147',
194: '5872829',
195: '5872219',
196: '5873271',
197: '5872954',
198: '5873215',
199: '5873218',
200: '5873211',
201: '5873357',
202: '5873220',
203: '5873214',
204: '5873524',
205: '5873525',
206: '6097864',
207: '6097865',
208: '6097868',
209: '6097861',
210: '6097854',
211: '6097862',
212: '6097867',
213: '6097857',
214: '6097871',
215: '6097869',
216: '6097872',
217: '6097890',
218: '6097883',
219: '6097879',
220: '6097886',
221: '6097878',
222: '6097885',
223: '6097882',
224: '6097896',
225: '6097892',
226: '6097900',
```

```
227: '6097906',
228: '6097912',
229: '6097924',
230: '6097897',
231: '6097927',
232: '6097923',
233: '6097926',
234: '7387665',
235: '6097930',
236: '7508714',
237: '7639821',
238: '7416497',
239: '7613168',
240: '7582513',
241: '7419401',
242: '7642452',
243: '7641814',
244: '7708350',
245: '7683246',
246: '7888598',
247: '7840196',
248: '7917481',
249: '7873810',
250: '7792354',
251: '7856746',
252: '7727698',
253: '7975487',
254: '8047965',
255: '8019310',
256: '8051076',
257: '8137203',
258: '8236590',
259: '8104778',
260: '8139233',
261: '8068514',
262: '8056040',
263: '8186956',
264: '8267077',
265: '8264200',
266: '8300237',
267: '8307815',
268: '8335452',
269: '8563753',
270: '8378220',
271: '8600499',
272: '9027622',
273: '8423772',
```

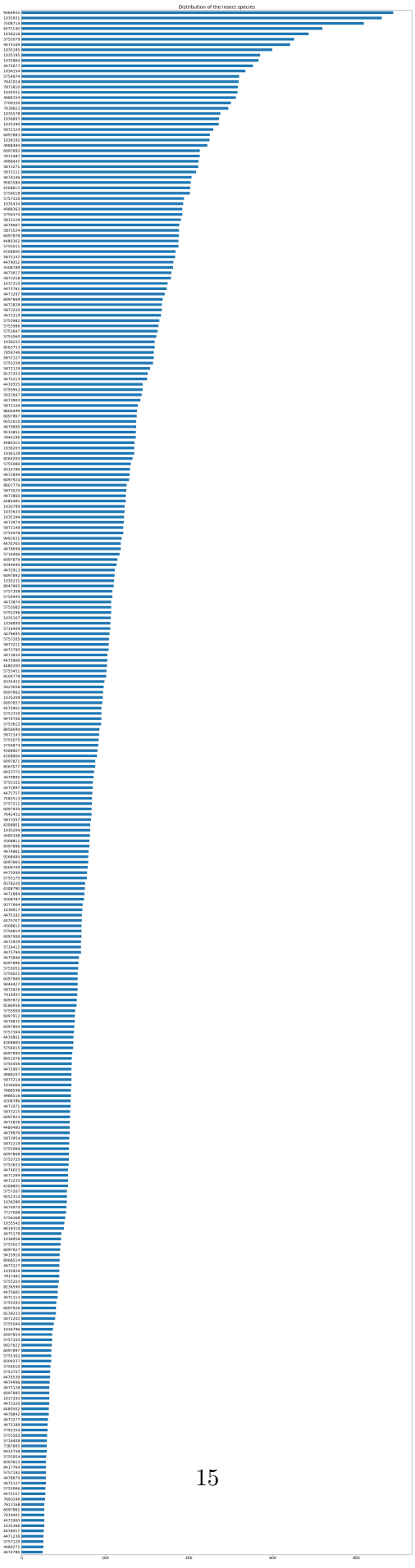
```
 274: '8417764',
 275: '8607776',
 276: '9206709',
 277: '9251010',
 278: '9268484',
 279: '9252314',
 280: '9346940',
 281: '9314786',
 282: '9444427',
 283: '9533851',
 284: '9479706',
 285: '9491931',
 286: '9415910',
 287: '9364935',
 288: '9414758',
 289: '9377664',
 290: '9581584'}
```

1.10 Exploratory Data Analysis (EDA)

[ ]:
```
# Calculate the number of images in the DataFrame.
# This is done by finding the length of the 'insect_gbif' column of the␣
 ↪DataFrame 'df'.
# The 'insect_gbif' column presumably contains unique identifiers for each␣
 ↪image.
num_images = len(df['insect_gbif'])

# Print the total number of images.
# This output will help in understanding the size of the dataset in terms of␣
 ↪images.
print('Number of images are:', num_images)

# Calculate the number of unique insect species.
# 'labels' is assumed to be a list or array containing the labels (species␣
 ↪names or identifiers) for each image.
# 'len(labels)' will give the total number of labels, which represents the␣
 ↪number of insect species.
no_labels = len(labels)

# Print the total number of unique insect species.
# This output provides an insight into the diversity of the dataset in terms of␣
 ↪different species it contains.
print('Number of insect species are:', no_labels)
```

```
Number of images are: 63364
Number of insect species are: 291
```

[ ]:
```
# Display the distribution of different insect species, identified by the␣
↪values in the 'insect_gbif' column of the DataFrame df.
bar = df['insect_gbif'].value_counts(ascending = True).plot.barh(figsize = (30,␣
↪120))
plt.title('Distribution of the insect species', fontsize = 20)
bar.tick_params(labelsize = 16)
plt.show()
```


Distribution of the insect species
0
200
400
600
800

[ ]:
```
# Count the occurrences of each unique value in the 'insect_gbif' column.
df['insect_gbif'].value_counts()
```

[ ]:
```
9364935    888
1035931    861
7508714    818
4475140    719
1036216    686
           …
4478057     53
4480271     52
4471238     52
5757120     52
4474780     50
Name: insect_gbif, Length: 291, dtype: int64
```

1.10.1 Load data using a Keras utility

[ ]:
```
batch_size = 128
img_height = 256
img_width = 256
image_size = (img_height, img_width)
seed = 123
shuffle_value = True
validation_split = 0.3
```

1.10.2 Split data to 70% training, 10% validation and 20% test.

Step 1: Split data

- 44,355 training
- 19,009 validation

[ ]:
```
# Split the data to train and validation
train_ds, val_ds = tf.keras.utils.image_dataset_from_directory(
  path,
  validation_split = validation_split,
  subset = 'both',
  seed = seed,
  image_size = image_size,
  color_mode = 'rgb',
  batch_size = batch_size,
  shuffle = shuffle_value
)
```

```
Found 63364 files belonging to 291 classes.
Using 44355 files for training.
Using 19009 files for validation.
```

Step 2: Split validation further:

- 6,336 validation
- 12,763 test

[ ]:
```
# Split validation to test and validation
val_batches = tf.data.experimental.cardinality(val_ds)
test_ds = val_ds.take((2 * val_batches) // 3)
val_ds = val_ds.skip((2 * val_batches) // 3)
```

You can find the class names in the class_names attribute on these datasets. These correspond to the directory names in alphabetical order.

[ ]:
```
class_names = train_ds.class_names
print(class_names)
```

```
['1035167', '1035185', '1035194', '1035195', '1035204', '1035208', '1035231',
'1035290', '1035366', '1035434', '1035542', '1035551', '1035578', '1035864',
'1035929', '1035931', '1036066', '1036128', '1036154', '1036192', '1036203',
'1036216', '1036255', '1036286', '1036789', '1036796', '1036893', '1036899',
'1036917', '1036958', '1037293', '1037319', '1037633', '4308786', '4308787',
'4308789', '4308790', '4308800', '4308801', '4308804', '4308805', '4308806',
'4308807', '4308811', '4308812', '4308815', '4470539', '4470555', '4470765',
'4470801', '4471071', '4471113', '4471202', '4471235', '4471238', '4471269',
'4472828', '4472849', '4472858', '4472884', '4472897', '4472900', '4472907',
'4472913', '4472929', '4473127', '4473277', '4473297', '4473319', '4473320',
'4473617', '4473795', '4473834', '4473860', '4473874', '4473903', '4473974',
'4474146', '4474169', '4474653', '4474780', '4474861', '4474974', '4474998',
'4475099', '4475127', '4475128', '4475140', '4475157', '4475179', '4475181',
'4475188', '4475213', '4475677', '4475685', '4475707', '4475757', '4475761',
'4475794', '4475846', '4475902', '4475908', '4476667', '4476832', '4478057',
'4478679', '4478695', '4478699', '4478842', '4479052', '4479870', '4479890',
'4479895', '4480200', '4480271', '4480302', '4480315', '4480348', '4480480',
'4480485', '4480502', '4988207', '4988354', '4988363', '4988447', '4988484',
'4988516', '5023047', '5023058', '5716406', '5716408', '5716409', '5716411',
'5753612', '5753653', '5753697', '5753720', '5753725', '5753757', '5754974',
'5755011', '5755027', '5755044', '5755051', '5755060', '5755066', '5755075',
'5755079', '5755080', '5755082', '5755175', '5755203', '5755293', '5755296',
'5755302', '5755322', '5755339', '5755451', '5755456', '5755562', '5755950',
'5755952', '5755954', '5755969', '5755978', '5755982', '5755986', '5756010',
'5756015', '5756018', '5756021', '5756368', '5756374', '5756814', '5756874',
'5756945', '5757120', '5757155', '5757192', '5757205', '5757207', '5757211',
'5757304', '5757308', '5757310', '5872109', '5872111', '5872119', '5872124',
'5872127', '5872129', '5872143', '5872147', '5872148', '5872219', '5872829',
'5872954', '5873211', '5873214', '5873215', '5873218', '5873220', '5873271',
```

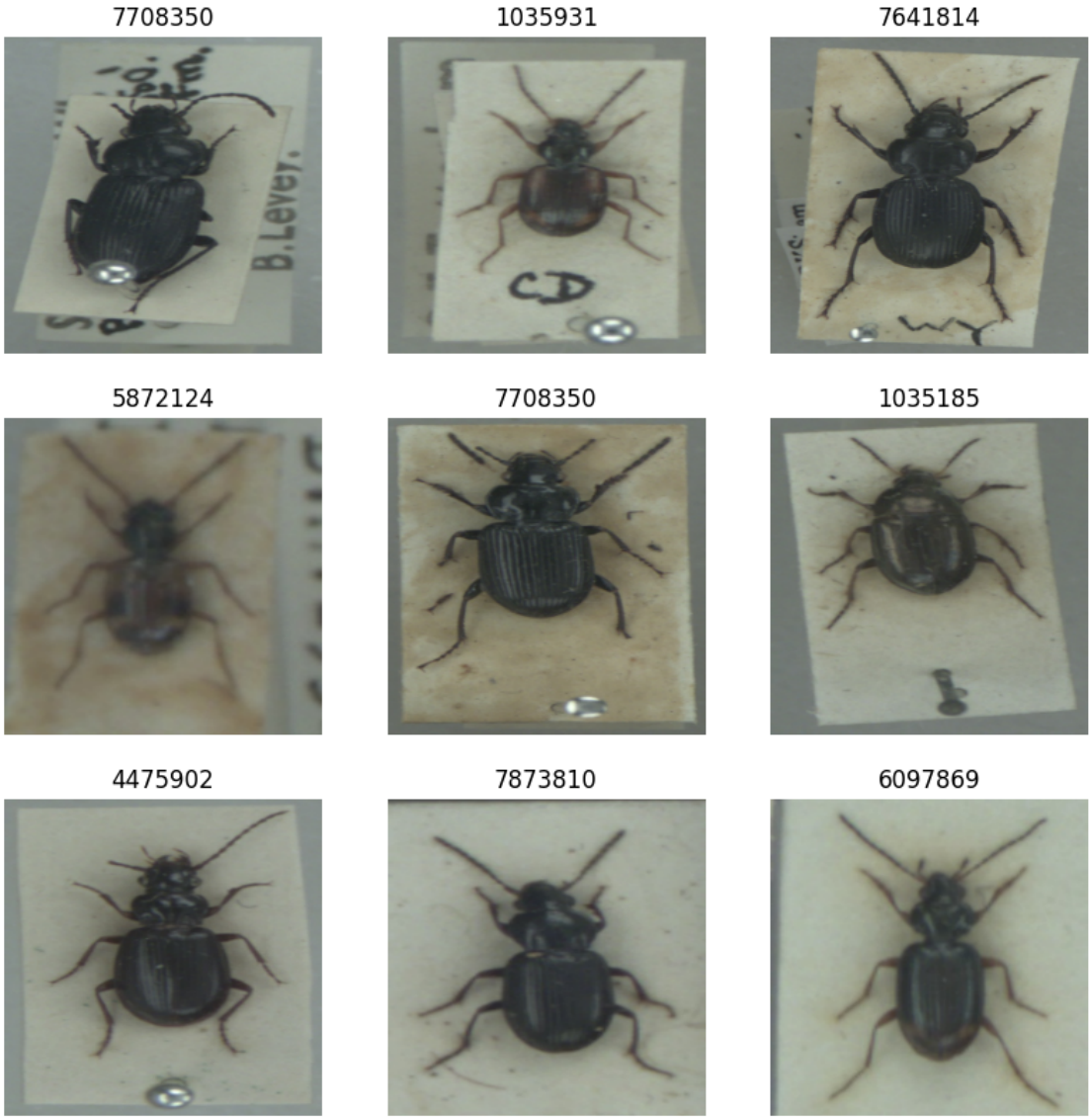
```
'5873357', '5873524', '5873525', '6097854', '6097857', '6097861', '6097862',
'6097864', '6097865', '6097867', '6097868', '6097869', '6097871', '6097872',
'6097878', '6097879', '6097882', '6097883', '6097885', '6097886', '6097890',
'6097892', '6097896', '6097897', '6097900', '6097906', '6097912', '6097923',
'6097924', '6097926', '6097927', '6097930', '7387665', '7416497', '7419401',
'7508714', '7582513', '7613168', '7639821', '7641814', '7642452', '7683246',
'7708350', '7727698', '7792354', '7840196', '7856746', '7873810', '7888598',
'7917481', '7975487', '8019310', '8047965', '8051076', '8056040', '8068514',
'8104778', '8137203', '8139233', '8186956', '8236590', '8264200', '8267077',
'8300237', '8307815', '8335452', '8378220', '8417764', '8423772', '8563753',
'8600499', '8607776', '9027622', '9206709', '9251010', '9252314', '9268484',
'9314786', '9346940', '9364935', '9377664', '9414758', '9415910', '9444427',
'9479706', '9491931', '9533851', '9581584']
```

1.11 Visualize the data

Verify we are able to visualize images.

Here are the first nine images from the training dataset:

[ ]:
```python
# Set up a figure for plotting images. 'figsize=(10, 10)' defines the size of the figure.
plt.figure(figsize=(10, 10))

# 'train_ds.take(1)' takes one batch of images and labels from the training dataset 'train_ds'.
# The for loop iterates through this single batch.
for images, labels in train_ds.take(1):
    # This nested for loop iterates through the first 9 images and their corresponding labels in the batch.
    for i in range(9):
        # 'plt.subplot(3, 3, i + 1)' creates a subplot in a 3x3 grid format.
        # 'i + 1' is the index of the subplot where the image will be displayed.
        ax = plt.subplot(3, 3, i + 1)

        # Display the i-th image in the batch.
        # 'images[i].numpy().astype("uint8")' converts the i-th image to a NumPy array with uint8 data type,
        # which is a standard for images.
        plt.imshow(images[i].numpy().astype('uint8'))

        # Set the title of the subplot to the label of the current image.
        # 'class_names[labels[i]]' translates the label index to a human-readable class name.
        plt.title(class_names[labels[i]])

        # Turn off the axis lines and labels for a cleaner image display.
        plt.axis("off")
```



Verify image and label batch shape

[ ]:
```
# Iterate over the batches of images and labels in the training dataset␣
↪'train_ds'.
for image_batch, labels_batch in train_ds:
    # Print the shape of the current batch of images.
    # 'image_batch.shape' will display the dimensions of the images in this␣
↪batch,
    # typically including the number of images, height, width, and color␣
↪channels (e.g., (32, 256, 256, 3) for 32 256x256 RGB images).
    print(image_batch.shape)
```

```
    # Print the shape of the current batch of labels.
    # 'labels_batch.shape' will display the dimensions of the labels array,
    # which typically includes the number of labels in the batch (e.g., (32,)␣
 ↪for 32 labels).
    print(labels_batch.shape)

    # Break the loop after processing the first batch.
    # This is used to only check the shapes of the images and labels in the␣
 ↪first batch without going through the entire dataset.
    break
```

```
(128, 256, 256, 3)
(128,)
```

The image_batch is a tensor of the shape (128, 256, 256, 3). This is a batch of 128 images of shape 256x256x3 (the last dimension refers to color channels RGB). The label_batch is a tensor of the shape (128,), these are corresponding labels to the 128 images.

1.12 Using image data augmentation

[ ]:
```
# Define a data augmentation pipeline using Keras's Sequential model.
# Data augmentation is a technique used to increase the diversity of your␣
 ↪training set
# by applying random but realistic transformations, such as flipping or␣
 ↪rotation.
data_augmentation = keras.Sequential(
    [
        # Add a 'RandomFlip' layer to the Sequential model.
        # This layer will randomly flip the input images along the horizontal␣
 ↪axis.
        # 'horizontal' indicates that the flip is along the horizontal axis.
        layers.RandomFlip('horizontal'),

        # Add a 'RandomRotation' layer to the Sequential model.
        # This layer will randomly rotate the input images during training.
        # '0.1' is the factor by which the images will be rotated.
        # It represents a fraction of 2 Pi (360 degrees), so 0.1 implies␣
 ↪rotation by up to 36 degrees.
        # A positive value means rotating counter-clockwise.
        layers.RandomRotation(0.1)
    ]
)
```

[ ]:
```
# Set up a figure for plotting images with a specific size.
# 'figsize=(10, 10)' defines the size of the figure as 10x10 inches.
plt.figure(figsize = (10, 10))
```

```
# Iterate over the first batch of images from the training dataset 'train_ds'.
# 'train_ds.take(1)' takes one batch from the dataset. The '_' is a placeholder␣
 ↪for the labels, which are ignored.
for images, _ in train_ds.take(1):
    # Iterate through the first 9 images in the batch.
    for i in range(9):
        # Apply the data augmentation defined in 'data_augmentation' to the␣
 ↪images.
        # This will randomly apply the transformations (flip, rotation) to each␣
 ↪image.
        augmented_images = data_augmentation(images)

        # Create a subplot in a 3x3 grid.
        # 'i + 1' is the position in the grid where the image will be plotted.
        ax = plt.subplot(3, 3, i + 1)

        # Display the first augmented image in each set of transformations.
        # '.numpy().astype('uint8')' converts the image data to a format␣
 ↪suitable for displaying with 'imshow'.
        plt.imshow(augmented_images[0].numpy().astype('uint8'))

        # Turn off the axis to make the plot cleaner.
        plt.axis('off')
```

1.13 Preprocess the data

1.13.1 Configure the dataset for performance

Create a Dataset that prefetches elements from this dataset.

Most dataset input pipelines should end with a call to prefetch. This allows later elements to be prepared while the current element is being processed. This often improves latency and throughput, at the cost of using additional memory to store prefetched elements. Since the value tf.data.AUTOTUNE is used, the buffer size is dynamically tuned.

[ ]:
```
# Apply `data_augmentation` to the training images.
# The 'map' function is used to apply a transformation to each element in the␣
↪dataset.
```

```
# Here, a lambda function is defined to apply the 'data_augmentation' to each␣
 ↪image in the dataset.
# The lambda function takes two arguments: 'img' (the image) and 'label' (its␣
 ↪label),
# and it returns the augmented image along with its label.
train_ds = train_ds.map(
    lambda img, label: (data_augmentation(img), label),
    # 'num_parallel_calls = tf.data.AUTOTUNE' optimizes the number of parallel␣
 ↪processing calls,
    # which can speed up the data loading and transformation process.
    num_parallel_calls=tf.data.AUTOTUNE,
)

# Prefetching the dataset.
# 'prefetch' allows the dataset to fetch batches in the background while the␣
 ↪model is being trained.
# This helps in reducing the time the model spends waiting for data, thus␣
 ↪improving efficiency.
# 'tf.data.AUTOTUNE' allows TensorFlow to automatically tune the amount of␣
 ↪prefetching dynamically at runtime.
train_ds = train_ds.prefetch(tf.data.AUTOTUNE)

# The same prefetching is applied to the validation dataset.
# This is important as it also helps in speeding up the validation process␣
 ↪during training.
val_ds = val_ds.prefetch(tf.data.AUTOTUNE)
```

1.14 Build a model

Building a neural network model for tasks such as image classification often involves a sequence of layers, each serving a specific purpose. Here's a more detailed explanation of the model building process as described in your statement:

1. **Starting with a Data Augmentation Preprocessor:**
   - The model begins with a data augmentation preprocessor. Data augmentation is a technique used to increase the diversity of the training dataset by applying random, yet realistic, transformations to the input images. This might include operations like rotating, zooming, flipping, etc.
   - Including data augmentation as the first layer in the model ensures that every image passed through the model during training is slightly altered, helping the model generalize better and reducing the risk of overfitting.
2. **Adding a Rescaling Layer:**
   - Following the data augmentation preprocessor, a Rescaling layer is added. This layer normalizes the pixel values in the image.
   - Typically, image data comes with pixel values in the range of 0-255. The Rescaling layer adjusts these values to a scale that's more suitable for neural network models, often between 0 and 1. For example, dividing each pixel value by 255 achieves this

normalization.

3. **Incorporating a Dropout Layer Before Final Classification:**
   - Before the final classification layer, a Dropout layer is incorporated into the model. Dropout is a regularization technique that randomly sets a fraction of the input units to 0 at each update during training time, which helps prevent overfitting.
   - The Dropout layer essentially forces the model to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
   - This layer is particularly important for complex models that are prone to overfitting, especially when trained on datasets of limited size.
4. **Final Classification Layer:**
   - The model concludes with a classification layer, which is typically a Dense layer with an activation function that corresponds to the problem at hand (e.g., a softmax activation for multi-class classification tasks).
   - The number of neurons in this final layer corresponds to the number of output classes. For binary classification, a single neuron with a sigmoid activation function might be used, while for multi-class classification problems, multiple neurons (equal to the number of classes) with a softmax activation function are common.

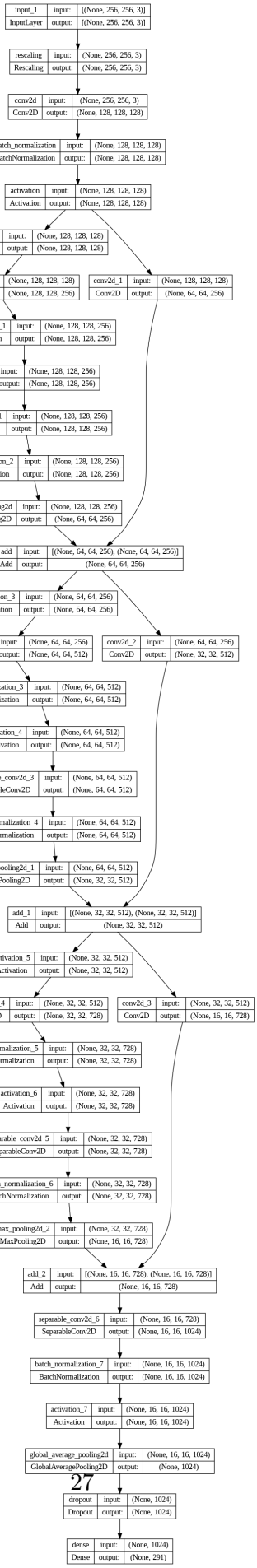
This structured approach in building the model – starting from data augmentation, normalizing data, adding regularization techniques, and ending with a proper classification layer – is designed to create a robust neural network capable of not only learning patterns in the training data but also generalizing well to new, unseen data.

[ ]:
```
def make_model(input_shape, num_classes):
    # Define the input layer with the specified shape (excluding batch size),
    # typically including height, width, and depth (e.g., channels in an image).
    inputs = keras.Input(shape = input_shape)

    # Entry block

    # Add a layer to rescale pixel values from [0, 255] to [0, 1],
    # which is a common practice to aid in neural network training.
    x = layers.Rescaling(1.0 / 255)(inputs)

    # Add a 2D convolution layer with 128 filters, a kernel size of 3x3,
    # stride of 2 (to reduce the spatial dimensions), and 'same' padding to␣
↪keep the spatial dimensions unchanged.
    x = layers.Conv2D(128, 3, strides = 2, padding = 'same')(x)

    # Normalize the outputs from the previous layer,
    # which helps in stabilizing and speeding up training.
    x = layers.BatchNormalization()(x)

    # Apply a ReLU activation function to introduce non-linearity.
    x = layers.Activation('relu')(x)

    # Save the current state (output) of the network to be used as a residual␣
↪connection later.
```

```
    previous_block_activation = x

    # Sequentially iterate over a set of filter sizes.
    # This loop creates multiple blocks of depthwise separable convolutions,
    # a computationally efficient alternative to regular convolutions.
    for size in [256, 512, 728]:
        x = layers.Activation('relu')(x)
        x = layers.SeparableConv2D(size, 3, padding = 'same')(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation('relu')(x)
        x = layers.SeparableConv2D(size, 3, padding = 'same')(x)
        x = layers.BatchNormalization()(x)

        # Apply max pooling to downsample the feature maps and reduce their␣
↪dimensions.
        x = layers.MaxPooling2D(3, strides = 2, padding = 'same')(x)

        # Project residual: adapt the residual tensor to have the same shape as␣
↪the current output.
        residual = layers.Conv2D(size, 1, strides = 2, padding =␣
↪'same')(previous_block_activation)

        # Add the residual tensor back to the output of the layer block,
        # enabling the flow of information and gradients through the network,
        # which helps in training deeper architectures.
        x = layers.add([x, residual])

        # Update the residual to the current state of the network.
        previous_block_activation = x

    # Add a final depthwise separable convolution layer with a large number of␣
↪filters.
    x = layers.SeparableConv2D(1024, 3, padding = 'same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)

    # Apply global average pooling to reduce each feature map to a single value,
    # which helps in reducing the number of parameters and computations in the␣
↪network.
    x = layers.GlobalAveragePooling2D()(x)

    # Decide the activation function and the number of units for the output␣
↪layer,
    # based on whether it's binary classification (2 classes) or multi-class.
    if num_classes == 2:
```

```
        activation = 'sigmoid'  # Use sigmoid for binary classification.
        units = 1
    else:
        activation = 'softmax'  # Use softmax for multi-class classification.
        units = num_classes

    # Add a Dropout layer to reduce overfitting by randomly setting a fraction␣
 ↪of the input units to 0.
    x = layers.Dropout(0.5)(x)

    # Define the output layer with the appropriate number of units and␣
 ↪activation function.
    outputs = layers.Dense(units, activation = activation)(x)

    # Construct the model with the specified inputs and outputs.
    return keras.Model(inputs, outputs)

# Instantiate the model with the specified input shape and number of classes.
model = make_model(input_shape=image_size + (3,), num_classes = 291)

# Generate a plot of the model architecture, showing the shape and connectivity␣
 ↪of each layer.
keras.utils.plot_model(model, show_shapes = True)
```

[ ]:


input_1 | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
rescaling | input: | (None, 256, 256, 3)
Rescaling | output: | (None, 256, 256, 3)
conv2d | input: | (None, 256, 256, 3)
Conv2D | output: | (None, 128, 128, 128)
batch_normalization | input: | (None, 128, 128, 128)
BatchNormalization | output: | (None, 128, 128, 128)
activation | input: | (None, 128, 128, 128)
Activation | output: | (None, 128, 128, 128)
activation_1 | input: | (None, 128, 128, 128)
Activation | output: | (None, 128, 128, 128)
separable_conv2d | input: | (None, 128, 128, 128)
SeparableConv2D | output: | (None, 128, 128, 256)
conv2d_1 | input: | (None, 128, 128, 128)
Conv2D | output: | (None, 64, 64, 256)
batch_normalization_1 | input: | (None, 128, 128, 256)
BatchNormalization | output: | (None, 128, 128, 256)
activation_2 | input: | (None, 128, 128, 256)
Activation | output: | (None, 128, 128, 256)
separable_conv2d_1 | input: | (None, 128, 128, 256)
SeparableConv2D | output: | (None, 128, 128, 256)
batch_normalization_2 | input: | (None, 128, 128, 256)
BatchNormalization | output: | (None, 128, 128, 256)
max_pooling2d | input: | (None, 128, 128, 256)
MaxPooling2D | output: | (None, 64, 64, 256)
add | input: | [(None, 64, 64, 256), (None, 64, 64, 256)]
Add | output: | (None, 64, 64, 256)
activation_3 | input: | (None, 64, 64, 256)
Activation | output: | (None, 64, 64, 256)
separable_conv2d_2 | input: | (None, 64, 64, 256)
SeparableConv2D | output: | (None, 64, 64, 512)
conv2d_2 | input: | (None, 64, 64, 256)
Conv2D | output: | (None, 32, 32, 512)
batch_normalization_3 | input: | (None, 64, 64, 512)
BatchNormalization | output: | (None, 64, 64, 512)
activation_4 | input: | (None, 64, 64, 512)
Activation | output: | (None, 64, 64, 512)
separable_conv2d_3 | input: | (None, 64, 64, 512)
SeparableConv2D | output: | (None, 64, 64, 512)
batch_normalization_4 | input: | (None, 64, 64, 512)
BatchNormalization | output: | (None, 64, 64, 512)
max_pooling2d_1 | input: | (None, 64, 64, 512)
MaxPooling2D | output: | (None, 32, 32, 512)
add_1 | input: | [(None, 32, 32, 512), (None, 32, 32, 512)]
Add | output: | (None, 32, 32, 512)
activation_5 | input: | (None, 32, 32, 512)
Activation | output: | (None, 32, 32, 512)
separable_conv2d_4 | input: | (None, 32, 32, 512)
SeparableConv2D | output: | (None, 32, 32, 728)
conv2d_3 | input: | (None, 32, 32, 512)
Conv2D | output: | (None, 16, 16, 728)
batch_normalization_5 | input: | (None, 32, 32, 728)
BatchNormalization | output: | (None, 32, 32, 728)
activation_6 | input: | (None, 32, 32, 728)
Activation | output: | (None, 32, 32, 728)
separable_conv2d_5 | input: | (None, 32, 32, 728)
SeparableConv2D | output: | (None, 32, 32, 728)
batch_normalization_6 | input: | (None, 32, 32, 728)
BatchNormalization | output: | (None, 32, 32, 728)
max_pooling2d_2 | input: | (None, 32, 32, 728)
MaxPooling2D | output: | (None, 16, 16, 728)
add_2 | input: | [(None, 16, 16, 728), (None, 16, 16, 728)]
Add | output: | (None, 16, 16, 728)
separable_conv2d_6 | input: | (None, 16, 16, 728)
SeparableConv2D | output: | (None, 16, 16, 1024)
batch_normalization_7 | input: | (None, 16, 16, 1024)
BatchNormalization | output: | (None, 16, 16, 1024)
activation_7 | input: | (None, 16, 16, 1024)
Activation | output: | (None, 16, 16, 1024)
global_average_pooling2d | input: | (None, 16, 16, 1024)
GlobalAveragePooling2D | output: | (None, 1024)
dropout | input: | (None, 1024)
Dropout | output: | (None, 1024)
dense | input: | (None, 1024)
Dense | output: | (None, 291)

[ ]:
```
# Prints a string summary of the network.
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 256, 256, 3)]     0

 rescaling (Rescaling)       (None, 256, 256, 3)       0

 conv2d (Conv2D)             (None, 128, 128, 128)     3584

 batch_normalization (BatchNormalization)  (None, 128, 128, 128)  512

 activation (Activation)     (None, 128, 128, 128)     0

 activation_1 (Activation)   (None, 128, 128, 128)     0

 separable_conv2d (SeparableConv2D)  (None, 128, 128, 256)  34176

 batch_normalization_1 (BatchNormalization)  (None, 128, 128, 256)  1024

 activation_2 (Activation)   (None, 128, 128, 256)     0

 separable_conv2d_1 (SeparableC  (None, 128, 128, 256)  68096
```

```
 onv2D)                         6)

 batch_normalization_2 (BatchNo  (None, 128, 128, 25  1024
['separable_conv2d_1[0][0]']
 rmalization)                   6)

 max_pooling2d (MaxPooling2D)   (None, 64, 64, 256)  0
['batch_normalization_2[0][0]']

 conv2d_1 (Conv2D)              (None, 64, 64, 256)  33024
['activation[0][0]']

 add (Add)                      (None, 64, 64, 256)  0
['max_pooling2d[0][0]',
'conv2d_1[0][0]']

 activation_3 (Activation)      (None, 64, 64, 256)  0           ['add[0][0]']

 separable_conv2d_2 (SeparableC  (None, 64, 64, 512)  133888
['activation_3[0][0]']
 onv2D)

 batch_normalization_3 (BatchNo  (None, 64, 64, 512)  2048
['separable_conv2d_2[0][0]']
 rmalization)

 activation_4 (Activation)      (None, 64, 64, 512)  0
['batch_normalization_3[0][0]']

 separable_conv2d_3 (SeparableC  (None, 64, 64, 512)  267264
['activation_4[0][0]']
 onv2D)

 batch_normalization_4 (BatchNo  (None, 64, 64, 512)  2048
['separable_conv2d_3[0][0]']
 rmalization)

 max_pooling2d_1 (MaxPooling2D)  (None, 32, 32, 512)  0
['batch_normalization_4[0][0]']

 conv2d_2 (Conv2D)              (None, 32, 32, 512)  131584      ['add[0][0]']

 add_1 (Add)                    (None, 32, 32, 512)  0
['max_pooling2d_1[0][0]',
'conv2d_2[0][0]']

 activation_5 (Activation)      (None, 32, 32, 512)  0           ['add_1[0][0]']
```

```
 separable_conv2d_4 (SeparableC  (None, 32, 32, 728)  378072
['activation_5[0][0]']
 onv2D)

 batch_normalization_5 (BatchNo  (None, 32, 32, 728)  2912
['separable_conv2d_4[0][0]']
 rmalization)

 activation_6 (Activation)      (None, 32, 32, 728)  0
['batch_normalization_5[0][0]']

 separable_conv2d_5 (SeparableC  (None, 32, 32, 728)  537264
['activation_6[0][0]']
 onv2D)

 batch_normalization_6 (BatchNo  (None, 32, 32, 728)  2912
['separable_conv2d_5[0][0]']
 rmalization)

 max_pooling2d_2 (MaxPooling2D)  (None, 16, 16, 728)  0
['batch_normalization_6[0][0]']

 conv2d_3 (Conv2D)              (None, 16, 16, 728)  373464      ['add_1[0][0]']

 add_2 (Add)                    (None, 16, 16, 728)  0
['max_pooling2d_2[0][0]',
'conv2d_3[0][0]']

 separable_conv2d_6 (SeparableC  (None, 16, 16, 1024  753048     ['add_2[0][0]']
 onv2D)                         )

 batch_normalization_7 (BatchNo  (None, 16, 16, 1024  4096
['separable_conv2d_6[0][0]']
 rmalization)                   )

 activation_7 (Activation)      (None, 16, 16, 1024  0
['batch_normalization_7[0][0]']
                                )

 global_average_pooling2d (Glob  (None, 1024)        0
['activation_7[0][0]']
 alAveragePooling2D)

 dropout (Dropout)              (None, 1024)         0
['global_average_pooling2d[0][0]'
                                                                  ]

 dense (Dense)                  (None, 291)          298275
```

```
['dropout[0][0]']

==========================================================================================
==================
Total params: 3,028,315
Trainable params: 3,020,027
Non-trainable params: 8,288
__________________________________________________________________________________________
__________________
```

1.15 Train the model

[ ]:
```
# Set the number of epochs, which are iterations over the entire dataset, to␣
 ↪train the model.
epochs = 100

# Define callbacks, which are utilities called at certain points during␣
 ↪training.
# Here, a ModelCheckpoint callback is used to save the model at the end of each␣
 ↪epoch.
callbacks = [
    keras.callbacks.ModelCheckpoint("save_at_{epoch}.keras")
]

# Compile the model with necessary configurations for training.
# - Optimizer: Adam, a popular optimization algorithm, with a learning rate of␣
 ↪1e-3.
# - Loss function: 'sparse_categorical_crossentropy', suitable for multi-class␣
 ↪classification
#   tasks where each class is mutually exclusive.
# - Metrics: Various metrics to monitor during training, including accuracy,␣
 ↪mean absolute error (mae),
#   categorical accuracy, sparse categorical accuracy, and top-k categorical␣
 ↪accuracy for k = 5.
model.compile(
    optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3),
    loss = 'sparse_categorical_crossentropy',
    metrics = [
        'accuracy',
        'mae',
        metrics.categorical_accuracy,
        metrics.sparse_categorical_accuracy,
        metrics.TopKCategoricalAccuracy(k = 5),
    ],
)

# Start training the model.
```

```
# - train_ds: The training dataset.
# - epochs: The number of epochs to train the model.
# - callbacks: The list of callbacks to apply during training.
# - validation_data: The dataset to use for validation.
# - use_multiprocessing: Enables the use of multiprocessing for data loading,␣
 ↪improving performance.
history = model.fit(
    train_ds,
    epochs = epochs,
    callbacks = callbacks,
    validation_data = val_ds,
    use_multiprocessing = True
)
```

```
Epoch 1/100
347/347 [==============================] - 994s 3s/step - loss: 3.8158 -
accuracy: 0.1710 - mae: 140.0289 - categorical_accuracy: 6.7636e-04 -
sparse_categorical_accuracy: 0.1710 - top_k_categorical_accuracy: 0.0075 -
val_loss: 8.6357 - val_accuracy: 0.0038 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0000e+00 - val_sparse_categorical_accuracy: 0.0038 -
val_top_k_categorical_accuracy: 0.0000e+00
Epoch 2/100
347/347 [==============================] - 140s 397ms/step - loss: 2.0647 -
accuracy: 0.4476 - mae: 140.0289 - categorical_accuracy: 0.0015 -
sparse_categorical_accuracy: 0.4476 - top_k_categorical_accuracy: 0.0166 -
val_loss: 7.3737 - val_accuracy: 0.0431 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0000e+00 - val_sparse_categorical_accuracy: 0.0431 -
val_top_k_categorical_accuracy: 0.0013
Epoch 3/100
347/347 [==============================] - 140s 398ms/step - loss: 1.3760 -
accuracy: 0.6079 - mae: 140.0289 - categorical_accuracy: 0.0025 -
sparse_categorical_accuracy: 0.6079 - top_k_categorical_accuracy: 0.0185 -
val_loss: 3.2970 - val_accuracy: 0.2929 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0000e+00 - val_sparse_categorical_accuracy: 0.2929 -
val_top_k_categorical_accuracy: 1.5780e-04
Epoch 4/100
347/347 [==============================] - 140s 398ms/step - loss: 1.0234 -
accuracy: 0.7017 - mae: 140.0289 - categorical_accuracy: 0.0028 -
sparse_categorical_accuracy: 0.7017 - top_k_categorical_accuracy: 0.0194 -
val_loss: 4.0350 - val_accuracy: 0.2642 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0000e+00 - val_sparse_categorical_accuracy: 0.2642 -
val_top_k_categorical_accuracy: 1.5780e-04
Epoch 5/100
347/347 [==============================] - 140s 398ms/step - loss: 0.8284 -
accuracy: 0.7538 - mae: 140.0289 - categorical_accuracy: 0.0026 -
sparse_categorical_accuracy: 0.7538 - top_k_categorical_accuracy: 0.0186 -
val_loss: 2.1609 - val_accuracy: 0.4750 - val_mae: 140.2227 -
```

```
val_categorical_accuracy: 4.7341e-04 - val_sparse_categorical_accuracy: 0.4750 -
val_top_k_categorical_accuracy: 0.0044
Epoch 6/100
347/347 [==============================] - 140s 398ms/step - loss: 0.6950 -
accuracy: 0.7898 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.7898 - top_k_categorical_accuracy: 0.0180 -
val_loss: 4.4692 - val_accuracy: 0.1687 - val_mae: 140.2227 -
val_categorical_accuracy: 9.4682e-04 - val_sparse_categorical_accuracy: 0.1687 -
val_top_k_categorical_accuracy: 0.0166
Epoch 7/100
347/347 [==============================] - 140s 398ms/step - loss: 0.6175 -
accuracy: 0.8120 - mae: 140.0289 - categorical_accuracy: 0.0031 -
sparse_categorical_accuracy: 0.8120 - top_k_categorical_accuracy: 0.0167 -
val_loss: 1.6107 - val_accuracy: 0.5733 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0076 - val_sparse_categorical_accuracy: 0.5733 -
val_top_k_categorical_accuracy: 0.0208
Epoch 8/100
347/347 [==============================] - 140s 398ms/step - loss: 0.5513 -
accuracy: 0.8318 - mae: 140.0289 - categorical_accuracy: 0.0031 -
sparse_categorical_accuracy: 0.8318 - top_k_categorical_accuracy: 0.0184 -
val_loss: 1.3768 - val_accuracy: 0.6109 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0038 - val_sparse_categorical_accuracy: 0.6109 -
val_top_k_categorical_accuracy: 0.0219
Epoch 9/100
347/347 [==============================] - 140s 398ms/step - loss: 0.4942 -
accuracy: 0.8486 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.8486 - top_k_categorical_accuracy: 0.0171 -
val_loss: 0.9862 - val_accuracy: 0.7141 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0028 - val_sparse_categorical_accuracy: 0.7141 -
val_top_k_categorical_accuracy: 0.0158
Epoch 10/100
347/347 [==============================] - 140s 398ms/step - loss: 0.4557 -
accuracy: 0.8596 - mae: 140.0289 - categorical_accuracy: 0.0032 -
sparse_categorical_accuracy: 0.8596 - top_k_categorical_accuracy: 0.0182 -
val_loss: 2.0350 - val_accuracy: 0.5018 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0000e+00 - val_sparse_categorical_accuracy: 0.5018 -
val_top_k_categorical_accuracy: 0.0052
Epoch 11/100
347/347 [==============================] - 140s 398ms/step - loss: 0.4277 -
accuracy: 0.8676 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.8676 - top_k_categorical_accuracy: 0.0177 -
val_loss: 3.7903 - val_accuracy: 0.3106 - val_mae: 140.2227 -
val_categorical_accuracy: 1.5780e-04 - val_sparse_categorical_accuracy: 0.3106 -
val_top_k_categorical_accuracy: 0.0046
Epoch 12/100
347/347 [==============================] - 140s 398ms/step - loss: 0.3924 -
accuracy: 0.8784 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.8784 - top_k_categorical_accuracy: 0.0169 -
```

```
val_loss: 1.9766 - val_accuracy: 0.5499 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0000e+00 - val_sparse_categorical_accuracy: 0.5499 -
val_top_k_categorical_accuracy: 0.0017
Epoch 13/100
347/347 [==============================] - 140s 398ms/step - loss: 0.3669 -
accuracy: 0.8861 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.8861 - top_k_categorical_accuracy: 0.0165 -
val_loss: 2.7621 - val_accuracy: 0.4620 - val_mae: 140.2227 -
val_categorical_accuracy: 1.5780e-04 - val_sparse_categorical_accuracy: 0.4620 -
val_top_k_categorical_accuracy: 0.0014
Epoch 14/100
347/347 [==============================] - 140s 398ms/step - loss: 0.3380 -
accuracy: 0.8946 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.8946 - top_k_categorical_accuracy: 0.0164 -
val_loss: 1.7136 - val_accuracy: 0.5843 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0161 - val_sparse_categorical_accuracy: 0.5843 -
val_top_k_categorical_accuracy: 0.0862
Epoch 15/100
347/347 [==============================] - 140s 398ms/step - loss: 0.3248 -
accuracy: 0.9000 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9000 - top_k_categorical_accuracy: 0.0163 -
val_loss: 2.3316 - val_accuracy: 0.4804 - val_mae: 140.2227 -
val_categorical_accuracy: 6.3121e-04 - val_sparse_categorical_accuracy: 0.4804 -
val_top_k_categorical_accuracy: 0.0049
Epoch 16/100
347/347 [==============================] - 140s 398ms/step - loss: 0.3049 -
accuracy: 0.9038 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9038 - top_k_categorical_accuracy: 0.0165 -
val_loss: 1.5400 - val_accuracy: 0.6169 - val_mae: 140.2227 -
val_categorical_accuracy: 7.8902e-04 - val_sparse_categorical_accuracy: 0.6169 -
val_top_k_categorical_accuracy: 0.0047
Epoch 17/100
347/347 [==============================] - 140s 398ms/step - loss: 0.2856 -
accuracy: 0.9089 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9089 - top_k_categorical_accuracy: 0.0170 -
val_loss: 1.8745 - val_accuracy: 0.5845 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0052 - val_sparse_categorical_accuracy: 0.5845 -
val_top_k_categorical_accuracy: 0.0260
Epoch 18/100
347/347 [==============================] - 140s 398ms/step - loss: 0.2783 -
accuracy: 0.9102 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9102 - top_k_categorical_accuracy: 0.0174 -
val_loss: 2.5462 - val_accuracy: 0.4865 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0060 - val_sparse_categorical_accuracy: 0.4865 -
val_top_k_categorical_accuracy: 0.0223
Epoch 19/100
347/347 [==============================] - 140s 398ms/step - loss: 0.2659 -
accuracy: 0.9143 - mae: 140.0289 - categorical_accuracy: 0.0030 -
```

```
sparse_categorical_accuracy: 0.9143 - top_k_categorical_accuracy: 0.0191 -
val_loss: 0.8588 - val_accuracy: 0.7620 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0013 - val_sparse_categorical_accuracy: 0.7620 -
val_top_k_categorical_accuracy: 0.0069
Epoch 20/100
347/347 [==============================] - 140s 397ms/step - loss: 0.2383 -
accuracy: 0.9226 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9226 - top_k_categorical_accuracy: 0.0156 -
val_loss: 1.6995 - val_accuracy: 0.6126 - val_mae: 140.2227 -
val_categorical_accuracy: 3.1561e-04 - val_sparse_categorical_accuracy: 0.6126 -
val_top_k_categorical_accuracy: 0.0041
Epoch 21/100
347/347 [==============================] - 140s 398ms/step - loss: 0.2383 -
accuracy: 0.9234 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9234 - top_k_categorical_accuracy: 0.0161 -
val_loss: 0.9758 - val_accuracy: 0.7313 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0016 - val_sparse_categorical_accuracy: 0.7313 -
val_top_k_categorical_accuracy: 0.0101
Epoch 22/100
347/347 [==============================] - 140s 398ms/step - loss: 0.2293 -
accuracy: 0.9265 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9265 - top_k_categorical_accuracy: 0.0168 -
val_loss: 2.3151 - val_accuracy: 0.5490 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0017 - val_sparse_categorical_accuracy: 0.5490 -
val_top_k_categorical_accuracy: 0.0066
Epoch 23/100
347/347 [==============================] - 140s 398ms/step - loss: 0.2162 -
accuracy: 0.9281 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9281 - top_k_categorical_accuracy: 0.0194 -
val_loss: 1.2000 - val_accuracy: 0.6975 - val_mae: 140.2227 -
val_categorical_accuracy: 6.3121e-04 - val_sparse_categorical_accuracy: 0.6975 -
val_top_k_categorical_accuracy: 0.0030
Epoch 24/100
347/347 [==============================] - 140s 398ms/step - loss: 0.2019 -
accuracy: 0.9340 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9340 - top_k_categorical_accuracy: 0.0177 -
val_loss: 1.7392 - val_accuracy: 0.6385 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0014 - val_sparse_categorical_accuracy: 0.6385 -
val_top_k_categorical_accuracy: 0.0084
Epoch 25/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1918 -
accuracy: 0.9375 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9375 - top_k_categorical_accuracy: 0.0187 -
val_loss: 1.7733 - val_accuracy: 0.6156 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0047 - val_sparse_categorical_accuracy: 0.6156 -
val_top_k_categorical_accuracy: 0.0245
Epoch 26/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1917 -
```

```
accuracy: 0.9377 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9377 - top_k_categorical_accuracy: 0.0176 -
val_loss: 1.1702 - val_accuracy: 0.7120 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0049 - val_sparse_categorical_accuracy: 0.7120 -
val_top_k_categorical_accuracy: 0.0202
Epoch 27/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1851 -
accuracy: 0.9384 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9384 - top_k_categorical_accuracy: 0.0171 -
val_loss: 1.1544 - val_accuracy: 0.6992 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0041 - val_sparse_categorical_accuracy: 0.6992 -
val_top_k_categorical_accuracy: 0.0265
Epoch 28/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1815 -
accuracy: 0.9403 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9403 - top_k_categorical_accuracy: 0.0178 -
val_loss: 2.2050 - val_accuracy: 0.5878 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0041 - val_sparse_categorical_accuracy: 0.5878 -
val_top_k_categorical_accuracy: 0.0140
Epoch 29/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1726 -
accuracy: 0.9431 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9431 - top_k_categorical_accuracy: 0.0184 -
val_loss: 1.4627 - val_accuracy: 0.6311 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0011 - val_sparse_categorical_accuracy: 0.6311 -
val_top_k_categorical_accuracy: 0.0114
Epoch 30/100
347/347 [==============================] - 140s 399ms/step - loss: 0.1570 -
accuracy: 0.9486 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9486 - top_k_categorical_accuracy: 0.0172 -
val_loss: 1.0187 - val_accuracy: 0.7371 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0047 - val_sparse_categorical_accuracy: 0.7371 -
val_top_k_categorical_accuracy: 0.0372
Epoch 31/100
347/347 [==============================] - 140s 399ms/step - loss: 0.1546 -
accuracy: 0.9488 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9488 - top_k_categorical_accuracy: 0.0176 -
val_loss: 3.7048 - val_accuracy: 0.4330 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0028 - val_sparse_categorical_accuracy: 0.4330 -
val_top_k_categorical_accuracy: 0.0142
Epoch 32/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1586 -
accuracy: 0.9472 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9472 - top_k_categorical_accuracy: 0.0168 -
val_loss: 2.5661 - val_accuracy: 0.5166 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0046 - val_sparse_categorical_accuracy: 0.5166 -
val_top_k_categorical_accuracy: 0.0230
Epoch 33/100
```

```
347/347 [==============================] - 140s 398ms/step - loss: 0.1466 -
accuracy: 0.9514 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9514 - top_k_categorical_accuracy: 0.0193 -
val_loss: 5.5437 - val_accuracy: 0.3618 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0095 - val_sparse_categorical_accuracy: 0.3618 -
val_top_k_categorical_accuracy: 0.0581
Epoch 34/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1355 -
accuracy: 0.9552 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9552 - top_k_categorical_accuracy: 0.0168 -
val_loss: 1.5469 - val_accuracy: 0.6715 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0032 - val_sparse_categorical_accuracy: 0.6715 -
val_top_k_categorical_accuracy: 0.0174
Epoch 35/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1445 -
accuracy: 0.9524 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9524 - top_k_categorical_accuracy: 0.0177 -
val_loss: 0.9455 - val_accuracy: 0.7756 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0062 - val_sparse_categorical_accuracy: 0.7756 -
val_top_k_categorical_accuracy: 0.0276
Epoch 36/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1296 -
accuracy: 0.9572 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9572 - top_k_categorical_accuracy: 0.0171 -
val_loss: 1.4697 - val_accuracy: 0.6969 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0028 - val_sparse_categorical_accuracy: 0.6969 -
val_top_k_categorical_accuracy: 0.0082
Epoch 37/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1311 -
accuracy: 0.9560 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9560 - top_k_categorical_accuracy: 0.0166 -
val_loss: 2.9454 - val_accuracy: 0.4900 - val_mae: 140.2227 -
val_categorical_accuracy: 3.1561e-04 - val_sparse_categorical_accuracy: 0.4900 -
val_top_k_categorical_accuracy: 0.0080
Epoch 38/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1269 -
accuracy: 0.9581 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9581 - top_k_categorical_accuracy: 0.0174 -
val_loss: 1.0202 - val_accuracy: 0.7807 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0036 - val_sparse_categorical_accuracy: 0.7807 -
val_top_k_categorical_accuracy: 0.0161
Epoch 39/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1225 -
accuracy: 0.9584 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9584 - top_k_categorical_accuracy: 0.0179 -
val_loss: 2.6470 - val_accuracy: 0.5312 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0014 - val_sparse_categorical_accuracy: 0.5312 -
val_top_k_categorical_accuracy: 0.0099
```

```
Epoch 40/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1191 -
accuracy: 0.9602 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9602 - top_k_categorical_accuracy: 0.0177 -
val_loss: 1.0517 - val_accuracy: 0.7510 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0028 - val_sparse_categorical_accuracy: 0.7510 -
val_top_k_categorical_accuracy: 0.0074
Epoch 41/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1115 -
accuracy: 0.9623 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9623 - top_k_categorical_accuracy: 0.0192 -
val_loss: 3.3222 - val_accuracy: 0.4605 - val_mae: 140.2227 -
val_categorical_accuracy: 6.3121e-04 - val_sparse_categorical_accuracy: 0.4605 -
val_top_k_categorical_accuracy: 0.0221
Epoch 42/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1154 -
accuracy: 0.9610 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9610 - top_k_categorical_accuracy: 0.0168 -
val_loss: 13.3867 - val_accuracy: 0.1436 - val_mae: 140.2227 -
val_categorical_accuracy: 6.3121e-04 - val_sparse_categorical_accuracy: 0.1436 -
val_top_k_categorical_accuracy: 0.0134
Epoch 43/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1092 -
accuracy: 0.9642 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9642 - top_k_categorical_accuracy: 0.0174 -
val_loss: 5.4129 - val_accuracy: 0.4486 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0043 - val_sparse_categorical_accuracy: 0.4486 -
val_top_k_categorical_accuracy: 0.0300
Epoch 44/100
347/347 [==============================] - 140s 399ms/step - loss: 0.1058 -
accuracy: 0.9635 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9635 - top_k_categorical_accuracy: 0.0156 -
val_loss: 2.7689 - val_accuracy: 0.5545 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0057 - val_sparse_categorical_accuracy: 0.5545 -
val_top_k_categorical_accuracy: 0.0374
Epoch 45/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1049 -
accuracy: 0.9642 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9642 - top_k_categorical_accuracy: 0.0165 -
val_loss: 2.2110 - val_accuracy: 0.6258 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0022 - val_sparse_categorical_accuracy: 0.6258 -
val_top_k_categorical_accuracy: 0.0098
Epoch 46/100
347/347 [==============================] - 140s 398ms/step - loss: 0.1002 -
accuracy: 0.9658 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9658 - top_k_categorical_accuracy: 0.0173 -
val_loss: 3.3835 - val_accuracy: 0.4813 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0055 - val_sparse_categorical_accuracy: 0.4813 -
```

```
val_top_k_categorical_accuracy: 0.0347
Epoch 47/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0911 -
accuracy: 0.9696 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9696 - top_k_categorical_accuracy: 0.0164 -
val_loss: 4.2497 - val_accuracy: 0.4854 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0038 - val_sparse_categorical_accuracy: 0.4854 -
val_top_k_categorical_accuracy: 0.0215
Epoch 48/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0942 -
accuracy: 0.9695 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9695 - top_k_categorical_accuracy: 0.0180 -
val_loss: 2.6834 - val_accuracy: 0.5825 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0062 - val_sparse_categorical_accuracy: 0.5825 -
val_top_k_categorical_accuracy: 0.0451
Epoch 49/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0920 -
accuracy: 0.9693 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9693 - top_k_categorical_accuracy: 0.0180 -
val_loss: 1.7180 - val_accuracy: 0.6555 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0076 - val_sparse_categorical_accuracy: 0.6555 -
val_top_k_categorical_accuracy: 0.0185
Epoch 50/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0929 -
accuracy: 0.9690 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9690 - top_k_categorical_accuracy: 0.0171 -
val_loss: 2.3215 - val_accuracy: 0.6421 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0030 - val_sparse_categorical_accuracy: 0.6421 -
val_top_k_categorical_accuracy: 0.0066
Epoch 51/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0901 -
accuracy: 0.9700 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9700 - top_k_categorical_accuracy: 0.0180 -
val_loss: 1.2141 - val_accuracy: 0.7581 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0027 - val_sparse_categorical_accuracy: 0.7581 -
val_top_k_categorical_accuracy: 0.0098
Epoch 52/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0855 -
accuracy: 0.9712 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9712 - top_k_categorical_accuracy: 0.0140 -
val_loss: 1.7243 - val_accuracy: 0.6823 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0036 - val_sparse_categorical_accuracy: 0.6823 -
val_top_k_categorical_accuracy: 0.0110
Epoch 53/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0885 -
accuracy: 0.9701 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9701 - top_k_categorical_accuracy: 0.0170 -
val_loss: 0.8783 - val_accuracy: 0.7956 - val_mae: 140.2227 -
```

```
val_categorical_accuracy: 0.0028 - val_sparse_categorical_accuracy: 0.7956 -
val_top_k_categorical_accuracy: 0.0093
Epoch 54/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0855 -
accuracy: 0.9709 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9709 - top_k_categorical_accuracy: 0.0165 -
val_loss: 0.8075 - val_accuracy: 0.8234 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0036 - val_sparse_categorical_accuracy: 0.8234 -
val_top_k_categorical_accuracy: 0.0107
Epoch 55/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0803 -
accuracy: 0.9729 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9729 - top_k_categorical_accuracy: 0.0170 -
val_loss: 5.7863 - val_accuracy: 0.4554 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0090 - val_sparse_categorical_accuracy: 0.4554 -
val_top_k_categorical_accuracy: 0.0723
Epoch 56/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0803 -
accuracy: 0.9736 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9736 - top_k_categorical_accuracy: 0.0181 -
val_loss: 3.2240 - val_accuracy: 0.5706 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0060 - val_sparse_categorical_accuracy: 0.5706 -
val_top_k_categorical_accuracy: 0.0320
Epoch 57/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0791 -
accuracy: 0.9741 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9741 - top_k_categorical_accuracy: 0.0174 -
val_loss: 2.0228 - val_accuracy: 0.6614 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0046 - val_sparse_categorical_accuracy: 0.6614 -
val_top_k_categorical_accuracy: 0.0245
Epoch 58/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0740 -
accuracy: 0.9752 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9752 - top_k_categorical_accuracy: 0.0171 -
val_loss: 1.3004 - val_accuracy: 0.7343 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0043 - val_sparse_categorical_accuracy: 0.7343 -
val_top_k_categorical_accuracy: 0.0273
Epoch 59/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0721 -
accuracy: 0.9752 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9752 - top_k_categorical_accuracy: 0.0163 -
val_loss: 5.7828 - val_accuracy: 0.4385 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0068 - val_sparse_categorical_accuracy: 0.4385 -
val_top_k_categorical_accuracy: 0.0379
Epoch 60/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0732 -
accuracy: 0.9759 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9759 - top_k_categorical_accuracy: 0.0167 -
```

```
val_loss: 2.5497 - val_accuracy: 0.6442 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0039 - val_sparse_categorical_accuracy: 0.6442 -
val_top_k_categorical_accuracy: 0.0180
Epoch 61/100
347/347 [==============================] - 140s 399ms/step - loss: 0.0744 -
accuracy: 0.9748 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9748 - top_k_categorical_accuracy: 0.0179 -
val_loss: 0.7894 - val_accuracy: 0.8214 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0027 - val_sparse_categorical_accuracy: 0.8214 -
val_top_k_categorical_accuracy: 0.0074
Epoch 62/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0692 -
accuracy: 0.9765 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9765 - top_k_categorical_accuracy: 0.0165 -
val_loss: 5.2401 - val_accuracy: 0.4832 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0047 - val_sparse_categorical_accuracy: 0.4832 -
val_top_k_categorical_accuracy: 0.0189
Epoch 63/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0747 -
accuracy: 0.9754 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9754 - top_k_categorical_accuracy: 0.0157 -
val_loss: 2.5326 - val_accuracy: 0.5888 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0022 - val_sparse_categorical_accuracy: 0.5888 -
val_top_k_categorical_accuracy: 0.0095
Epoch 64/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0675 -
accuracy: 0.9779 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9779 - top_k_categorical_accuracy: 0.0167 -
val_loss: 0.7530 - val_accuracy: 0.8267 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0036 - val_sparse_categorical_accuracy: 0.8267 -
val_top_k_categorical_accuracy: 0.0131
Epoch 65/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0693 -
accuracy: 0.9775 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9775 - top_k_categorical_accuracy: 0.0195 -
val_loss: 1.2652 - val_accuracy: 0.7652 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0035 - val_sparse_categorical_accuracy: 0.7652 -
val_top_k_categorical_accuracy: 0.0118
Epoch 66/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0750 -
accuracy: 0.9743 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9743 - top_k_categorical_accuracy: 0.0178 -
val_loss: 1.4201 - val_accuracy: 0.7319 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0076 - val_sparse_categorical_accuracy: 0.7319 -
val_top_k_categorical_accuracy: 0.0798
Epoch 67/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0623 -
accuracy: 0.9788 - mae: 140.0289 - categorical_accuracy: 0.0029 -
```

```
sparse_categorical_accuracy: 0.9788 - top_k_categorical_accuracy: 0.0179 -
val_loss: 1.4509 - val_accuracy: 0.7478 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0028 - val_sparse_categorical_accuracy: 0.7478 -
val_top_k_categorical_accuracy: 0.0069
Epoch 68/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0632 -
accuracy: 0.9781 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9781 - top_k_categorical_accuracy: 0.0167 -
val_loss: 2.8072 - val_accuracy: 0.5919 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0046 - val_sparse_categorical_accuracy: 0.5919 -
val_top_k_categorical_accuracy: 0.0245
Epoch 69/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0695 -
accuracy: 0.9770 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9770 - top_k_categorical_accuracy: 0.0168 -
val_loss: 1.1518 - val_accuracy: 0.7655 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0016 - val_sparse_categorical_accuracy: 0.7655 -
val_top_k_categorical_accuracy: 0.0095
Epoch 70/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0623 -
accuracy: 0.9791 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9791 - top_k_categorical_accuracy: 0.0178 -
val_loss: 0.9352 - val_accuracy: 0.7930 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0035 - val_sparse_categorical_accuracy: 0.7930 -
val_top_k_categorical_accuracy: 0.0156
Epoch 71/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0608 -
accuracy: 0.9796 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9796 - top_k_categorical_accuracy: 0.0162 -
val_loss: 1.9885 - val_accuracy: 0.6842 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0014 - val_sparse_categorical_accuracy: 0.6842 -
val_top_k_categorical_accuracy: 0.0063
Epoch 72/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0586 -
accuracy: 0.9805 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9805 - top_k_categorical_accuracy: 0.0146 -
val_loss: 3.9196 - val_accuracy: 0.5220 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0054 - val_sparse_categorical_accuracy: 0.5220 -
val_top_k_categorical_accuracy: 0.0260
Epoch 73/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0622 -
accuracy: 0.9793 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9793 - top_k_categorical_accuracy: 0.0168 -
val_loss: 0.9034 - val_accuracy: 0.8270 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0039 - val_sparse_categorical_accuracy: 0.8270 -
val_top_k_categorical_accuracy: 0.0136
Epoch 74/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0597 -
```

```
accuracy: 0.9800 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9800 - top_k_categorical_accuracy: 0.0164 -
val_loss: 1.6441 - val_accuracy: 0.7212 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0032 - val_sparse_categorical_accuracy: 0.7212 -
val_top_k_categorical_accuracy: 0.0079
Epoch 75/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0581 -
accuracy: 0.9800 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9800 - top_k_categorical_accuracy: 0.0176 -
val_loss: 2.0984 - val_accuracy: 0.6786 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0039 - val_sparse_categorical_accuracy: 0.6786 -
val_top_k_categorical_accuracy: 0.0122
Epoch 76/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0564 -
accuracy: 0.9809 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9809 - top_k_categorical_accuracy: 0.0161 -
val_loss: 2.4559 - val_accuracy: 0.6412 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0103 - val_sparse_categorical_accuracy: 0.6412 -
val_top_k_categorical_accuracy: 0.0715
Epoch 77/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0573 -
accuracy: 0.9811 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9811 - top_k_categorical_accuracy: 0.0160 -
val_loss: 3.3756 - val_accuracy: 0.5965 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0039 - val_sparse_categorical_accuracy: 0.5965 -
val_top_k_categorical_accuracy: 0.0114
Epoch 78/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0539 -
accuracy: 0.9816 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9816 - top_k_categorical_accuracy: 0.0163 -
val_loss: 1.7194 - val_accuracy: 0.7074 - val_mae: 140.2227 -
val_categorical_accuracy: 9.4682e-04 - val_sparse_categorical_accuracy: 0.7074 -
val_top_k_categorical_accuracy: 0.0030
Epoch 79/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0523 -
accuracy: 0.9826 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9826 - top_k_categorical_accuracy: 0.0173 -
val_loss: 1.1380 - val_accuracy: 0.7814 - val_mae: 140.2227 -
val_categorical_accuracy: 9.4682e-04 - val_sparse_categorical_accuracy: 0.7814 -
val_top_k_categorical_accuracy: 0.0041
Epoch 80/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0584 -
accuracy: 0.9807 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9807 - top_k_categorical_accuracy: 0.0157 -
val_loss: 0.9586 - val_accuracy: 0.8102 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0021 - val_sparse_categorical_accuracy: 0.8102 -
val_top_k_categorical_accuracy: 0.0055
Epoch 81/100
```

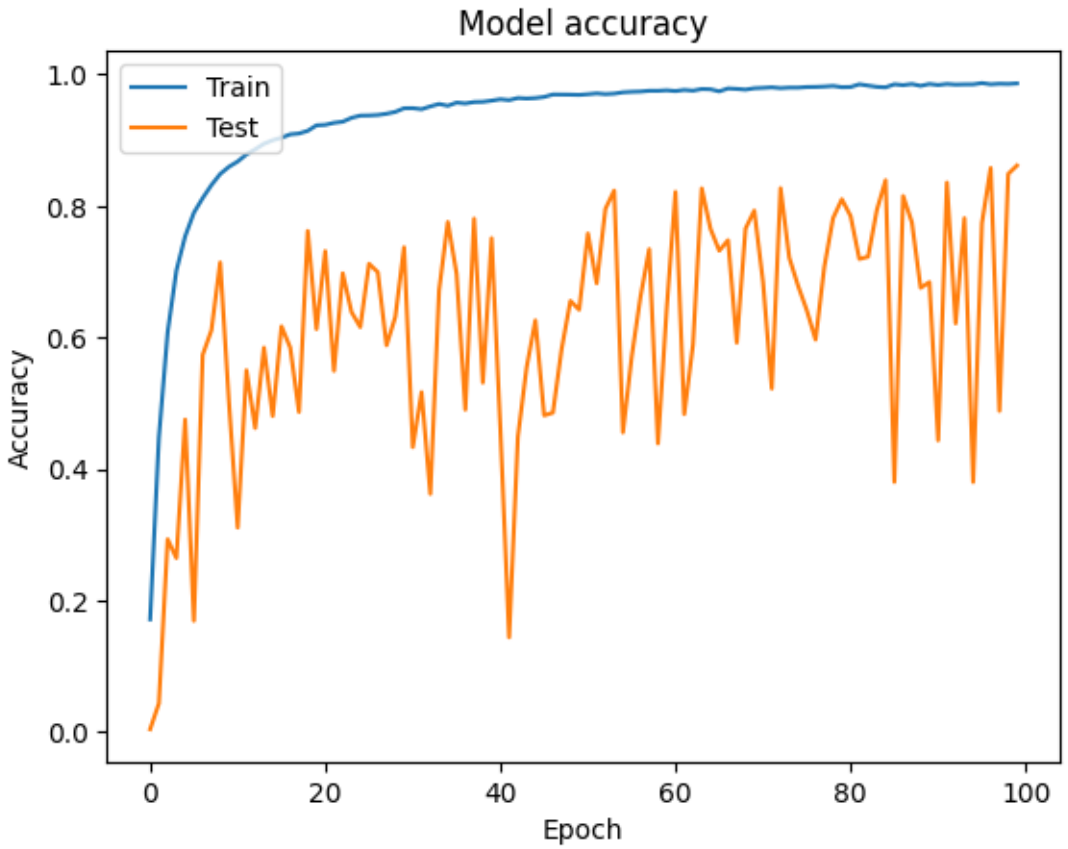
```
347/347 [==============================] - 140s 398ms/step - loss: 0.0584 -
accuracy: 0.9810 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9810 - top_k_categorical_accuracy: 0.0189 -
val_loss: 1.1762 - val_accuracy: 0.7846 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0033 - val_sparse_categorical_accuracy: 0.7846 -
val_top_k_categorical_accuracy: 0.0098
Epoch 82/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0462 -
accuracy: 0.9851 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9851 - top_k_categorical_accuracy: 0.0163 -
val_loss: 1.5053 - val_accuracy: 0.7193 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0028 - val_sparse_categorical_accuracy: 0.7193 -
val_top_k_categorical_accuracy: 0.0084
Epoch 83/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0514 -
accuracy: 0.9831 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9831 - top_k_categorical_accuracy: 0.0186 -
val_loss: 1.7444 - val_accuracy: 0.7227 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0039 - val_sparse_categorical_accuracy: 0.7227 -
val_top_k_categorical_accuracy: 0.0096
Epoch 84/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0566 -
accuracy: 0.9812 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9812 - top_k_categorical_accuracy: 0.0163 -
val_loss: 1.0882 - val_accuracy: 0.7939 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0041 - val_sparse_categorical_accuracy: 0.7939 -
val_top_k_categorical_accuracy: 0.0101
Epoch 85/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0559 -
accuracy: 0.9805 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9805 - top_k_categorical_accuracy: 0.0152 -
val_loss: 0.8798 - val_accuracy: 0.8392 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0044 - val_sparse_categorical_accuracy: 0.8392 -
val_top_k_categorical_accuracy: 0.0305
Epoch 86/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0443 -
accuracy: 0.9851 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9851 - top_k_categorical_accuracy: 0.0158 -
val_loss: 5.7045 - val_accuracy: 0.3800 - val_mae: 140.2227 -
val_categorical_accuracy: 6.3121e-04 - val_sparse_categorical_accuracy: 0.3800 -
val_top_k_categorical_accuracy: 0.0122
Epoch 87/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0487 -
accuracy: 0.9838 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9838 - top_k_categorical_accuracy: 0.0163 -
val_loss: 0.9715 - val_accuracy: 0.8152 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0115 - val_sparse_categorical_accuracy: 0.8152 -
val_top_k_categorical_accuracy: 0.0413
```

```
Epoch 88/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0447 -
accuracy: 0.9855 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9855 - top_k_categorical_accuracy: 0.0179 -
val_loss: 1.3322 - val_accuracy: 0.7750 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0033 - val_sparse_categorical_accuracy: 0.7750 -
val_top_k_categorical_accuracy: 0.0140
Epoch 89/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0525 -
accuracy: 0.9827 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9827 - top_k_categorical_accuracy: 0.0183 -
val_loss: 2.1638 - val_accuracy: 0.6754 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0035 - val_sparse_categorical_accuracy: 0.6754 -
val_top_k_categorical_accuracy: 0.0096
Epoch 90/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0437 -
accuracy: 0.9856 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9856 - top_k_categorical_accuracy: 0.0153 -
val_loss: 2.0249 - val_accuracy: 0.6841 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0000e+00 - val_sparse_categorical_accuracy: 0.6841 -
val_top_k_categorical_accuracy: 6.3121e-04
Epoch 91/100
347/347 [==============================] - 140s 399ms/step - loss: 0.0466 -
accuracy: 0.9840 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9840 - top_k_categorical_accuracy: 0.0158 -
val_loss: 6.3998 - val_accuracy: 0.4433 - val_mae: 140.2227 -
val_categorical_accuracy: 1.5780e-04 - val_sparse_categorical_accuracy: 0.4433 -
val_top_k_categorical_accuracy: 0.0036
Epoch 92/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0452 -
accuracy: 0.9856 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9856 - top_k_categorical_accuracy: 0.0148 -
val_loss: 0.8135 - val_accuracy: 0.8356 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0039 - val_sparse_categorical_accuracy: 0.8356 -
val_top_k_categorical_accuracy: 0.0133
Epoch 93/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0469 -
accuracy: 0.9846 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9846 - top_k_categorical_accuracy: 0.0172 -
val_loss: 2.4196 - val_accuracy: 0.6213 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0025 - val_sparse_categorical_accuracy: 0.6213 -
val_top_k_categorical_accuracy: 0.0118
Epoch 94/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0447 -
accuracy: 0.9850 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9850 - top_k_categorical_accuracy: 0.0171 -
val_loss: 1.0530 - val_accuracy: 0.7814 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0017 - val_sparse_categorical_accuracy: 0.7814 -
```

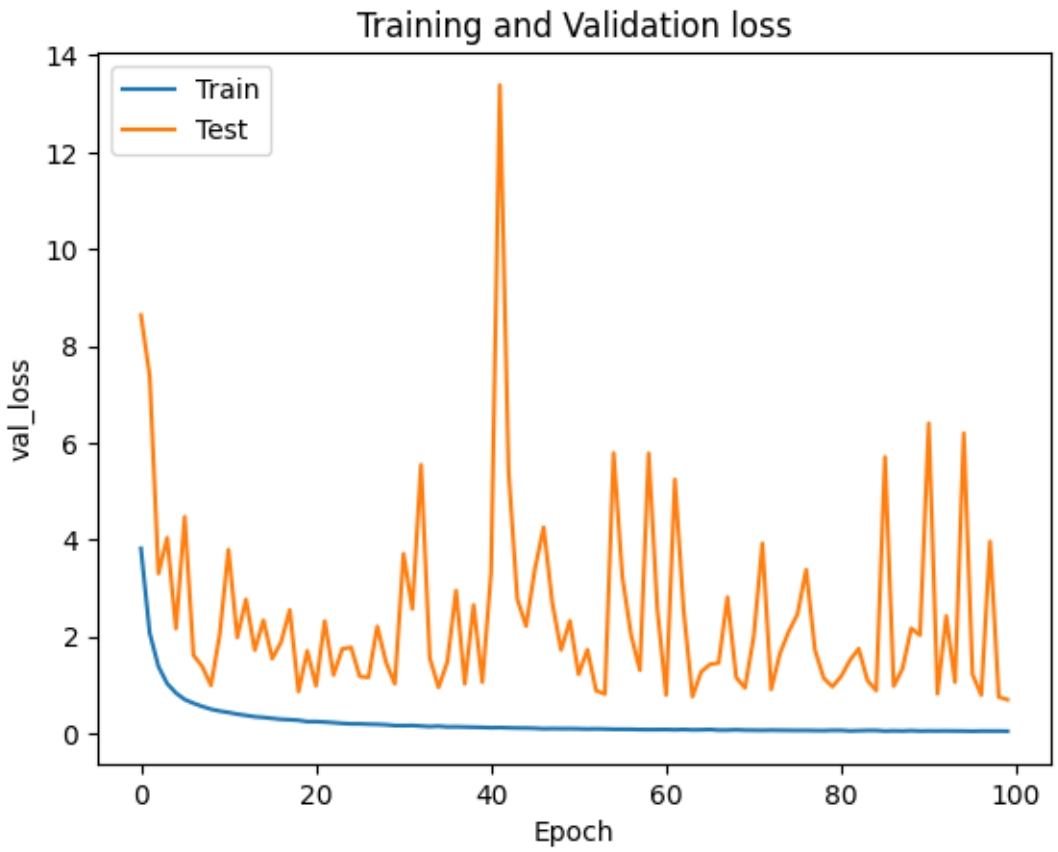
```
val_top_k_categorical_accuracy: 0.0057
Epoch 95/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0438 -
accuracy: 0.9850 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9850 - top_k_categorical_accuracy: 0.0203 -
val_loss: 6.1960 - val_accuracy: 0.3798 - val_mae: 140.2227 -
val_categorical_accuracy: 9.4682e-04 - val_sparse_categorical_accuracy: 0.3798 -
val_top_k_categorical_accuracy: 0.0049
Epoch 96/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0395 -
accuracy: 0.9869 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9869 - top_k_categorical_accuracy: 0.0143 -
val_loss: 1.2139 - val_accuracy: 0.7737 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0060 - val_sparse_categorical_accuracy: 0.7737 -
val_top_k_categorical_accuracy: 0.0494
Epoch 97/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0441 -
accuracy: 0.9851 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9851 - top_k_categorical_accuracy: 0.0156 -
val_loss: 0.7874 - val_accuracy: 0.8578 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0047 - val_sparse_categorical_accuracy: 0.8578 -
val_top_k_categorical_accuracy: 0.0298
Epoch 98/100
347/347 [==============================] - 140s 398ms/step - loss: 0.0429 -
accuracy: 0.9859 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9859 - top_k_categorical_accuracy: 0.0153 -
val_loss: 3.9584 - val_accuracy: 0.4881 - val_mae: 140.2227 -
val_categorical_accuracy: 3.1561e-04 - val_sparse_categorical_accuracy: 0.4881 -
val_top_k_categorical_accuracy: 0.0022
Epoch 99/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0429 -
accuracy: 0.9856 - mae: 140.0289 - categorical_accuracy: 0.0029 -
sparse_categorical_accuracy: 0.9856 - top_k_categorical_accuracy: 0.0198 -
val_loss: 0.7475 - val_accuracy: 0.8487 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0046 - val_sparse_categorical_accuracy: 0.8487 -
val_top_k_categorical_accuracy: 0.0211
Epoch 100/100
347/347 [==============================] - 140s 397ms/step - loss: 0.0404 -
accuracy: 0.9864 - mae: 140.0289 - categorical_accuracy: 0.0030 -
sparse_categorical_accuracy: 0.9864 - top_k_categorical_accuracy: 0.0146 -
val_loss: 0.6910 - val_accuracy: 0.8614 - val_mae: 140.2227 -
val_categorical_accuracy: 0.0054 - val_sparse_categorical_accuracy: 0.8614 -
val_top_k_categorical_accuracy: 0.0249
```

[ ]:
```python
# Plot training & validation accuracy values
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
```

```
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Test'], loc = 'upper left')
plt.show()
```



[ ]:
```
# Plot training & validation loss values
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Test'], loc = 'upper left')
plt.show()
```



1.16 Evaluating the model

[ ]:
```
# Evaluate the model's performance on the training dataset.
# 'model.evaluate' returns the loss value & metrics values for the model in␣
 ↪test mode.
# Computation is done in batches (hence, the dataset passed is 'train_ds').
train_score = model.evaluate(train_ds)

# Print the training loss.
# 'train_score[0]' refers to the first element in the result returned by 'model.
 ↪evaluate', which is the loss.
print('Train loss:', train_score[0])

# Print the training accuracy.
# 'train_score[1]' refers to the second element, which in this case is the␣
 ↪accuracy metric.
print('Train accuracy:', train_score[1])

# Print the Mean Absolute Error (MAE) for the training data.
```

```
# 'train_score[2]' refers to the third element, which is the MAE metric in this␣
 ↪setup.
print('Train mae:', train_score[2])
```

```
347/347 [==============================] - 61s 169ms/step - loss: 0.3338 -
accuracy: 0.9080 - mae: 140.0289 - categorical_accuracy: 0.0034 -
sparse_categorical_accuracy: 0.9080 - top_k_categorical_accuracy: 0.0225
Train loss: 0.33379465341567993
Train accuracy: 0.9080374240875244
Train mae: 140.0289306640625
```

[ ]:
```
# Evaluate the model's performance on the test dataset.
# 'model.evaluate' computes the loss and metrics for the model in test mode.
# The test dataset 'test_ds' is passed to this function, and evaluation is done␣
 ↪in batches.
test_score = model.evaluate(test_ds)

# Print the test loss.
# 'test_score[0]' refers to the first element in the array returned by 'model.
 ↪evaluate', which is the loss on the test dataset.
print('Test loss:', test_score[0])

# Print the test accuracy.
# 'test_score[1]' refers to the second element in the array, which in this case␣
 ↪is the accuracy metric on the test dataset.
print('Test accuracy:', test_score[1])

# Print the Mean Absolute Error (MAE) for the test data.
# 'test_score[2]' refers to the third element in the array, which is the MAE␣
 ↪metric in this setup.
print('Test mae:', test_score[2])
```

```
99/99 [==============================] - 12s 116ms/step - loss: 0.7054 -
accuracy: 0.8579 - mae: 137.8542 - categorical_accuracy: 0.0047 -
sparse_categorical_accuracy: 0.8579 - top_k_categorical_accuracy: 0.0250
Test loss: 0.7053909301757812
Test accuracy: 0.857875645160675
Test mae: 137.85415649414062
```

1.17 Save Model

[ ]:
```
# Save the model to file
filename = 'finalized_model.pkl'

with open(filename, 'wb') as file:
  pickle.dump(model, file)
```

1.18 Load the Model

[ ]:
```
with open(filename, 'rb') as file:
  loaded_model = pickle.load(file)
```

1.19 Conclusions

In this individual research project, I successfully developed a Convolutional Neural Network (CNN) for Insect Image Classification, achieving noteworthy results. The model attained a high training accuracy of 98.64% with a loss of 4.04%, and on the test set, it demonstrated an accuracy of 85.79% with a loss of 70.74%. These results are encouraging, yet a detailed analysis of the learning curves and dataset composition suggests areas for further refinement.

During the evaluation of learning curves, I observed that the model neither underfit nor overfit the training data. Underfitting typically manifests as a persistently flat training loss, or a continuously decreasing loss until the end of training, neither of which was present in this case. Overfitting, indicated by a continuously decreasing training loss or an initial decrease in validation loss followed by an increase, was also not observed. However, the learning curve for the validation loss exhibited fluctuating and somewhat noisy movements in comparison to the training loss. This pattern indicates that the validation dataset may not have been fully representative, potentially affecting my ability to accurately assess the model's generalization.

The dataset composition played a crucial role in this observation. From the 63,364 insect images available from Kaggle (https://www.kaggle.com/kmldas/insect-identification-from-habitus-images), I allocated 43,355 images for training, 6,336 for validation, and 12,672 for testing. The relatively smaller size and possibly limited diversity of the validation set might have constrained its capacity to thoroughly challenge the model, thereby influencing the evaluation results.

In conclusion, while the achieved results demonstrate the viability of using CNNs for classifying insect images, they also underscore the importance of a well-balanced and representative dataset for both training and validation. This project highlights the need for a larger and more varied set of images to ensure a robust evaluation of the model's generalization abilities. With a more comprehensive dataset encompassing a wider array of insect species, I am confident that the effectiveness of CNNs in this domain can be more conclusively established. This endeavor illustrates both the potential and the challenges in the field of image classification, paving the way for further research and development.