CNN Image Classification Laboration

Images used in this laboration are from CIFAR10. The CIFAR10 dataset contains 60,000 32x32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.

Your task is to make a classifier, using a convolutional neural network, that can correctly classify each image into the correct class.

Complete the code flagged throughout the elaboration and **answer all the questions in the notebook**.

In [1]:
```
# Setups
# Automatically reload modules when changed
%reload_ext autoreload
%autoreload 2
```

Part 1: Convolutions

In the next sections you will familiarize yourself with 2D convolutions.

1.1 What is a convolution?

To understand a bit more about convolutions, we will first test the convolution function in `scipy` using a number of classical filters.

Convolve the image with Gaussian filter, a Sobel X filter, and a Sobel Y filter, using the function `convolve2d` in `signal` from scipy (see the documentation for more details).

In a CNN, many filters are applied in each layer, and the filter coefficients are learned through back propagation (which is in contrast to traditional image processing, where the filters are designed by an expert).

Run the cell below to define a Gaussian filter and a Sobel X and Y filters.

In [2]:
```
from scipy import signal
import numpy as np

# Get a test image
from scipy import datasets
image = datasets.ascent()

# Define a help function for creating a Gaussian filter
def matlab_style_gauss2D(shape=(3,3),sigma=0.5):
    """
    2D gaussian mask - should give the same result as MATLAB's
```

```
    fspecial('gaussian',[shape],[sigma])
    """
    m,n = [(ss-1.)/2. for ss in shape]
    y,x = np.ogrid[-m:m+1,-n:n+1]
    h = np.exp( -(x*x + y*y) / (2.*sigma*sigma) )
    h[ h < np.finfo(h.dtype).eps*h.max() ] = 0
    sumh = h.sum()
    if sumh != 0:
        h /= sumh
    return h

# Create Gaussian filter with certain size and standard deviation
gaussFilter = matlab_style_gauss2D((15,15),4)
# A Gaussian filter is used for smoothing the image.
# In this case, the filter size is (15, 15) with a standard deviation of 4.

# Mainly used to remove noise from an image while preserving the overall structu
# A smoothing effect is achieved by reducing high frequency noise in an image by

# Define filter kernels for SobelX and Sobely
sobelX = np.array([[ 1, 0,  -1],
                    [2, 0, -2],
                    [1, 0, -1]])
# Used to detect vertical edges in an image
# because the pixel variation in the horizontal direction is greatest at vertica
# Vertical edges will be more visible in the output image.

sobelY = np.array([[ 1, 2,  1],
                    [0, 0, 0],
                    [-1, -2, -1]])
# Used to detect horizontal edges in an image
# because the pixel variation in the vertical direction is greatest at horizonta
# Horizontal edges will be more visible in the output image.
```

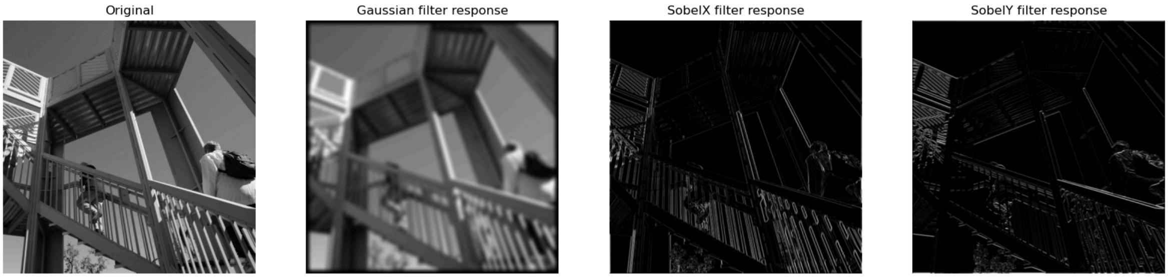
In [3]:

```
# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------
# Perform convolution using the function 'convolve2d' for the different filters

filterResponseGauss = signal.convolve2d(image, gaussFilter, mode='full', boundar
filterResponseSobelX = signal.convolve2d(image, sobelX, mode='full', boundary='f
filterResponseSobelY = signal.convolve2d(image, sobelY, mode='full', boundary='f

# ============================================
```

In [4]:

```
import matplotlib.pyplot as plt

# Show filter responses
fig, (ax_orig, ax_filt1, ax_filt2, ax_filt3) = plt.subplots(1, 4, figsize=(20, 6
ax_orig.imshow(image, cmap='gray')
ax_orig.set_title('Original')
ax_orig.set_axis_off()
ax_filt1.imshow(np.absolute(filterResponseGauss), cmap='gray')
ax_filt1.set_title('Gaussian filter response')
ax_filt1.set_axis_off()
ax_filt2.imshow(np.absolute(filterResponseSobelX), cmap='gray')
ax_filt2.set_title('SobelX filter response')
ax_filt2.set_axis_off()
ax_filt3.imshow(np.absolute(filterResponseSobelY), cmap='gray')
```

```
ax_filt3.set_title('SobelY filter response')
ax_filt3.set_axis_off()
```



1.2 Understanding convolutions

Questions

1. What do the 3 different filters (Gaussian, SobelX, SobelY) do to the original image?
2. What is the size of the original image? How many channels does it have? How many channels does a color image normally have?
3. What is the size of the different filters?
4. What is the size of the filter response if mode 'same' is used for the convolution ?
5. What is the size of the filter response if mode 'valid' is used for the convolution? How does the size of the valid filter response depend on the size of the filter?
6. Why are 'valid' convolutions a problem for CNNs with many layers?

Answers

1. Gaussian filter would blur the image, suppress high-frequency noise, smooth the overall. Vertical edges (e.g. left and right boundaries of an object) are shown as bright lines after SobelX filter. Horizontal edges (such as the top and bottom boundaries of an object) are displayed as bright lines after SobelY.
2. The channel of the grey image is 1, while the number for colorful image should be 3 generally.
3. The size of Gaussian filter is (15, 15), others are (3, 3)
4. We take Gaussian filter for example, the size of filter response is (512, 512)
5. The size of filter response for Gaussian filter decreases to (498, 498), while the figures for the other two are (510, 510). The reason is the center of the filter must always lie within the valid region of the input image, when the filter is close to the edge of the image, some of its regions go beyond the input boundaries (without padding), so the number of slides actually available is

reduced. As for how the size of filter affects the size of response in the 'valid' mode, the rule is like:

$$OutputHeight = H - K_h + 1$$

$$OutputWidth = W - K_w + 1$$

where, H and W are input height and width respectively, and K_h and K_w are filter height and width.

6. Because there is no any padding for the response, the size of response would decrease after layers. It may decreases to (1, 1), which cannot go through another layer. Except that, as mentioned reason, it would lose the edge information, affecting the performance of the model.

In [5]:

```
# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------
# Your code for checking sizes of image and filter responses
print(f"The dimension of image is: {image.shape}")
filterResponse_test = signal.convolve2d(image, gaussFilter, mode='same', boundar
print(f"The dimension of filter response of Gaussian filter with parameter mode=
filterResponse_valid_test1 = signal.convolve2d(image, gaussFilter, mode='valid',
print(f"The dimension of filter response of Gaussian filter with parameter mode=

filterResponse_valid_test2 = signal.convolve2d(image, sobelX, mode='valid', boun
print(f"The dimension of filter response of SobelX filter with parameter mode='v

filterResponse_valid_test3 = signal.convolve2d(image, sobelY, mode='valid', boun
print(f"The dimension of filter response of SobelY filter with parameter mode='v

# ==============================================
```

```
The dimension of image is: (512, 512)
The dimension of filter response of Gaussian filter with parameter mode='same' is
(512, 512)
The dimension of filter response of Gaussian filter with parameter mode='valid' i
s (498, 498)
The dimension of filter response of SobelX filter with parameter mode='valid' is
(510, 510)
The dimension of filter response of SobelY filter with parameter mode='valid' is
(510, 510)
```

Part 2: Get a graphics card

Skip the next cell if you run on the CPU.

If your computer has a dedicated graphics card and you would like to use it, we need to make sure that our script can see the graphics card that will be used. The graphics cards will perform all the time consuming calculations in every training iteration.

In [6]:

```
import os
import warnings
```

```python
# Ignore FutureWarning from numpy
warnings.simplefilter(action='ignore', category=FutureWarning)

import tensorflow as tf

os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"

# The GPU id to use, usually either "0" or "1";
os.environ["CUDA_VISIBLE_DEVICES"]="0"

# This sets the GPU to allocate memory only as needed
physical_devices = tf.config.experimental.list_physical_devices('GPU')
if len(physical_devices) != 0:
    tf.config.experimental.set_memory_growth(physical_devices[0], True)
    print("Running on GPU")
else:
    print('No GPU available.')
```

```
No GPU available.
```

How fast is the graphics card?

Questions

7. Why are the filters used for a color image of size 7 x 7 x 3, and not 7 x 7 ?
8. What operation is performed by the 'Conv2D' layer? Is it a standard 2D convolution, as performed by the function signal.convolve2d we just tested?
9. Pretend that everyone is using an Nvidia RTX 3090 graphics card, how many CUDA cores does it have? How much memory does the graphics card have?
10. How much memory does the graphics card have?
11. What is stored in the GPU memory while training a CNN?
12. Do you think that a graphics card, compared to the CPU, is equally faster for convolving a batch of 1,000 images, compared to convolving a batch of 3 images? Motivate your answer.

Answers

7. Colorful images need channels to represent the color information. (7, 7) represents grey images.
8. The Conv2D layer performs a two-dimensional Cross-Correlation rather than a mathematically strict Convolution. The difference between the two is whether the filter flips.
9. Number of CUDA cores: 10496, graphics memory capacity 24GB.
10. If it is Nvidia RTX 3060, it is 24GB.
11. Weights of model, the present batch of data and labels, gradients and activations.

12. I don't think so. When the batch size is 3, many GPU cores remain unoccupied, meaning it cannot fully leverage parallel computing. Additionally, transferring data to the GPU takes time. In this case, I don't believe the GPU is necessarily faster than the CPU for a task with a batch size of 3.

Part 3: Dataset

In the following section you will load the CIFAR10 dataset, check few samples, perform some preprocessing on the images and the labels, and split the data into training, validation and testing.

3.1 Load the dataset

Run the following section to load the CIFAR10 data, take a total of 10.000 training/validation samples and 2000 testing samples.

In [7]:
```
from keras.datasets import cifar10
import numpy as np

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship'

# Download CIFAR train and test data
(X, Y), (Xtest, Ytest) = cifar10.load_data()

print("Training/validation images have size {} and labels have size {} ".format(
print("Test images have size {} and labels have size {} \n ".format(Xtest.shape,

# Reduce the number of images for training/validation and testing to 10000 and 2
# to reduce processing time for this elaboration.
X = X[0:10000]
Y = Y[0:10000]

Xtest = Xtest[0:2000]
Ytest = Ytest[0:2000]

Ytestint = Ytest

print("Reduced training/validation images have size %s and labels have size %s "
print("Reduced test images have size %s and labels have size %s \n" % (Xtest.sha

# Check that we have some training examples from each class
for i in range(10):
    print("Number of training/validation examples for class {} is {}" .format(i,
```

```
Training/validation images have size (50000, 32, 32, 3) and labels have size (500
00, 1)
Test images have size (10000, 32, 32, 3) and labels have size (10000, 1)

Reduced training/validation images have size (10000, 32, 32, 3) and labels have s
ize (10000, 1)
Reduced test images have size (2000, 32, 32, 3) and labels have size (2000, 1)

Number of training/validation examples for class 0 is 1005
Number of training/validation examples for class 1 is 974
Number of training/validation examples for class 2 is 1032
Number of training/validation examples for class 3 is 1016
Number of training/validation examples for class 4 is 999
Number of training/validation examples for class 5 is 937
Number of training/validation examples for class 6 is 1030
Number of training/validation examples for class 7 is 1001
Number of training/validation examples for class 8 is 1025
Number of training/validation examples for class 9 is 981
```

Lets look at some of the training examples, this cell is already finished. You will see different examples every time you run the cell.

In [8]:

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(12,4))
for i in range(18):
    idx = np.random.randint(7500)
    label = Y[idx,0]

    plt.subplot(3,6,i+1)
    plt.tight_layout()
    plt.imshow(X[idx])
    plt.title("Class: {} ({})".format(label, classes[label]))
    plt.axis('off')
plt.show()
```

3.2 Split data into training, validation and testing

Split your data (X, Y) into training (Xtrain, Ytrain) and validation (Xval, Yval), so that we have training, validation and test datasets (as in the previous laboration).

We use the `train_test_split` function from scikit learn (see the documentation for more details) to obtain 25% validation set.

In [9]:
```
from sklearn.model_selection import train_test_split

# -------------------------------------------
# === Your code here ========================
# -------------------------------------------

# split the original dataset into 70% Training and 30% Temp
Xtrain, X_temp, Ytrain, Y_temp = train_test_split(X, Y, test_size=0.3, random_st



# Print the size of training data, validation data and test data
Xval, Xtest, Yval, Ytest = train_test_split(X_temp, Y_temp, test_size=0.5, randc

# ===========================================
print(f"Training set size: {len(Xtrain)}")
print(f"Validation set size: {len(Xval)}")
print(f"Test set size: {len(Xtest)}")
```

```
Training set size: 7000
Validation set size: 1500
Test set size: 1500
```

3.3 Image Preprocessing

Lets perform some preprocessing. The images are stored as uint8, i.e. 8 bit unsigned integers, but need to be converted to 32 bit floats. We also make sure that the range is -1 to 1, instead of 0 - 255.

In [10]:
```
# Convert datatype for Xtrain, Xval, Xtest, to float32
Xtrain = Xtrain.astype('float32')
Xval = Xval.astype('float32')
Xtest = Xtest.astype('float32')

# Change range of pixel values to [-1,1]
Xtrain = Xtrain / 127.5 - 1
Xval = Xval / 127.5 - 1
Xtest = Xtest / 127.5 - 1
```

3.4 Label preprocessing

The labels (Y) need to be converted from e.g. '4' to "hot encoded", i.e. to a vector of type [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] . We use the `to_categorical` function in Keras (see the documentation for details on how to use it)

In [11]:
```
from tensorflow.keras.utils import to_categorical

# Print shapes before converting the labels
print('Ytrain has size {}.'.format(Ytrain.shape))
print('Yval has size {}.'.format(Yval.shape))
print('Ytest has size {}.'.format(Ytest.shape))


# -------------------------------------------
# === Your code here ========================
```

```
# ---------------------------------------------
num_classes = Ytrain.max() + 1
# Your code for converting Ytrain, Yval, Ytest to categorical
Ytrain = to_categorical(Ytrain, num_classes)
Yval = to_categorical(Yval, num_classes)
Ytest = to_categorical(Ytest, num_classes)


# Print shapes after converting the labels
print('After one-hot encoding:')
print('Ytrain has size {}.'.format(Ytrain.shape))
print('Yval has size {}.'.format(Yval.shape))
print('Ytest has size {}.'.format(Ytest.shape))

# =============================================
```

```
Ytrain has size (7000, 1).
Yval has size (1500, 1).
Ytest has size (1500, 1).
After one-hot encoding:
Ytrain has size (7000, 10).
Yval has size (1500, 10).
Ytest has size (1500, 10).
```

Part 4: 2D CNN

In the following sections you will build a 2D CNN model and will train it to perform classification on the CIFAR10 dataset.

4.1 Build CNN model

Start by implementing the `build_CNN` function in the `utilities.py` file. Below you can find the specifications on how your `build_CNN` function should build the model:

- Each convolutional layer is composed by: `2D convolution` -> `batch normalization` -> `max pooling`.
- The `2D convolution` uses a 3 x 3 kernel size, padding='same' and a number of starting filter that is an input to the `build_CNN` function. The number of filters doubles with each convolutional layer (e.g. 32, 64, 128, etc.)
- The max pooling layers should have a pool size of 2 x 2.
- After the convolutional layers comes a flatten layer, followed by a number of intermediate dense layers.
- The number of nodes in the intermediate dense layers before the final dense layer is an input to the `build_CNN` function. The intermediate dense layers use `relu` activation functions and each is followed by `batch normalization`.
- The final dense layer should have 10 nodes (=the number of classes in this elaboration) and `softmax` activation.

Here are some relevant functions that you should use in `build_CNN`. For a complete list of functions and their definitions see the keras documentation:

- `model.add()`, adds a layer to the network;
- `Dense()`, a dense network layer. See the documentation what are the input options and outputs of the `Dense()` function.
- `Conv2D()` performs 2D convolutions with a number of filters with a certain size (e.g. 3 x 3) (see documentation).
- `BatchNormalization()`, perform batch normalization (see documentation).
- `MaxPooling2D()`, saves the max for a given pool size, results in down sampling (see documentation).
- `Flatten()`, flatten a multi-channel tensor into a long vector (see documentation).
- `model.compile()`, compiles the model. You can set the input metrics=['accuracy'] to print the classification accuracy during the training.
- cost and loss functions: check the documentation and chose a loss function for binary classification.

To get more information in model compile, training and evaluation see the relevant documentation.

Here you can start with the `Adam` optimizer when compiling the model.

Use the following cell to test your `build_CNN` utility function. Remember to import a relevant cost function for multi-class classification from keras.losses which relates to how many classes you have.

In [12]:
```
# import utilities
from utilities import build_CNN

# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------

# import a suitable loss function from keras.losses and use as input to the buil
from tensorflow.keras.losses import CategoricalCrossentropy

input_shape = Xtest[1].shape
# print(f"input shape is {input_shape}")
# Build a DNN model following the specifications above
model = build_CNN(input_shape, CategoricalCrossentropy())

# =============================================
```

4.2 Train 2D CNN

Time to train the CNN!

Start with a model with 2 convolutional layers where the first layer has have 16 filters, and with no intermediate dense layers.

Set the training parameters, build the model and run the training.

Use the following training parameters:

- `batch_size=20`
- `epochs=20`
- `learning_rate=0.01`

Relevant functions:

- `build_CNN`, the function that you defined in the `utilities.py` file.
- `model.fit()`, train the model with some training data (see documentation).
- `model.evaluate()`, apply the trained model to some test data (see documentation).

2 convolutional layers, no intermediate dense layers

In [13]:

```
# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

# Setup some training parameters
batch_size = 20
epochs = 20
input_shape = input_shape
learning_rate = 0.01

# Build model
model1 = build_CNN(input_shape=input_shape,
                   loss=CategoricalCrossentropy(),
                   learning_rate=learning_rate)

# Train the model  using training data and validation data
history1 = model1.fit(Xtrain,
                      Ytrain,
                      validation_data=(Xval, Yval),
                      epochs=epochs,
                      batch_size=batch_size)

# ============================================
```

```
Epoch 1/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.3036 - loss: 2.3233 - val_
accuracy: 0.4260 - val_loss: 1.6277
Epoch 2/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.4883 - loss: 1.4987 - val_
accuracy: 0.4947 - val_loss: 1.4661
Epoch 3/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.5517 - loss: 1.2593 - val_
accuracy: 0.5360 - val_loss: 1.3732
Epoch 4/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.6114 - loss: 1.0976 - val_
accuracy: 0.5327 - val_loss: 1.3617
Epoch 5/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.6569 - loss: 0.9893 - val_
accuracy: 0.5733 - val_loss: 1.3044
Epoch 6/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.6944 - loss: 0.8657 - val_
accuracy: 0.5720 - val_loss: 1.2792
Epoch 7/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.7251 - loss: 0.7944 - val_
accuracy: 0.5713 - val_loss: 1.3343
Epoch 8/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.7488 - loss: 0.7169 - val_
accuracy: 0.5747 - val_loss: 1.3115
Epoch 9/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.7838 - loss: 0.6344 - val_
accuracy: 0.5740 - val_loss: 1.3795
Epoch 10/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8199 - loss: 0.5612 - val_
accuracy: 0.5827 - val_loss: 1.3669
Epoch 11/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8238 - loss: 0.5298 - val_
accuracy: 0.5673 - val_loss: 1.5017
Epoch 12/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8333 - loss: 0.4866 - val_
accuracy: 0.5553 - val_loss: 1.5359
Epoch 13/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8630 - loss: 0.4086 - val_
accuracy: 0.5720 - val_loss: 1.5260
Epoch 14/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8732 - loss: 0.3907 - val_
accuracy: 0.5673 - val_loss: 1.6172
Epoch 15/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8885 - loss: 0.3380 - val_
accuracy: 0.5560 - val_loss: 1.6895
Epoch 16/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8989 - loss: 0.3023 - val_
accuracy: 0.5547 - val_loss: 1.7747
Epoch 17/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9072 - loss: 0.2913 - val_
accuracy: 0.5540 - val_loss: 1.7405
Epoch 18/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9124 - loss: 0.2692 - val_
accuracy: 0.5580 - val_loss: 1.8184
Epoch 19/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9340 - loss: 0.2213 - val_
accuracy: 0.5560 - val_loss: 1.8703
Epoch 20/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9532 - loss: 0.1787 - val_
accuracy: 0.5627 - val_loss: 1.9115
```

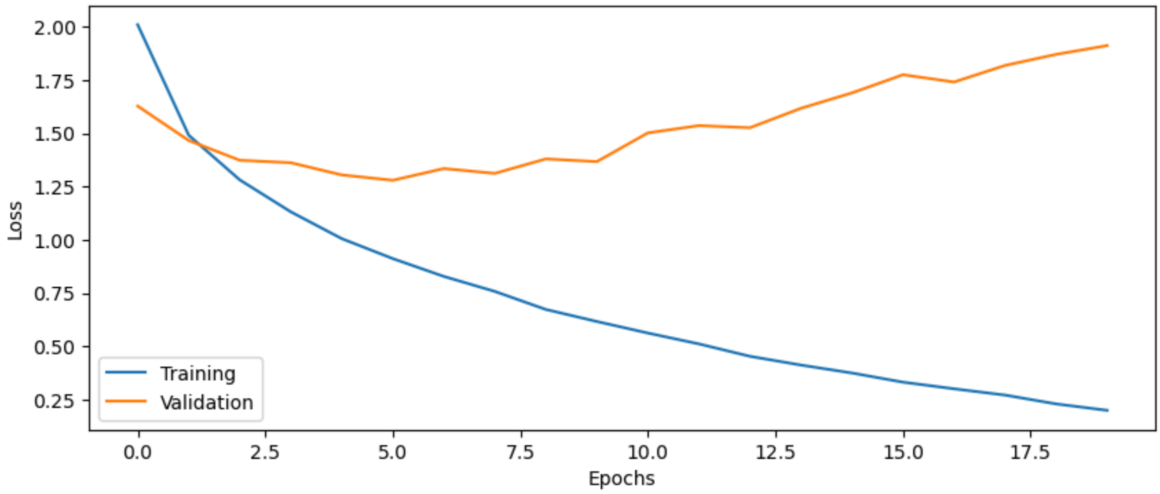
In [14]:

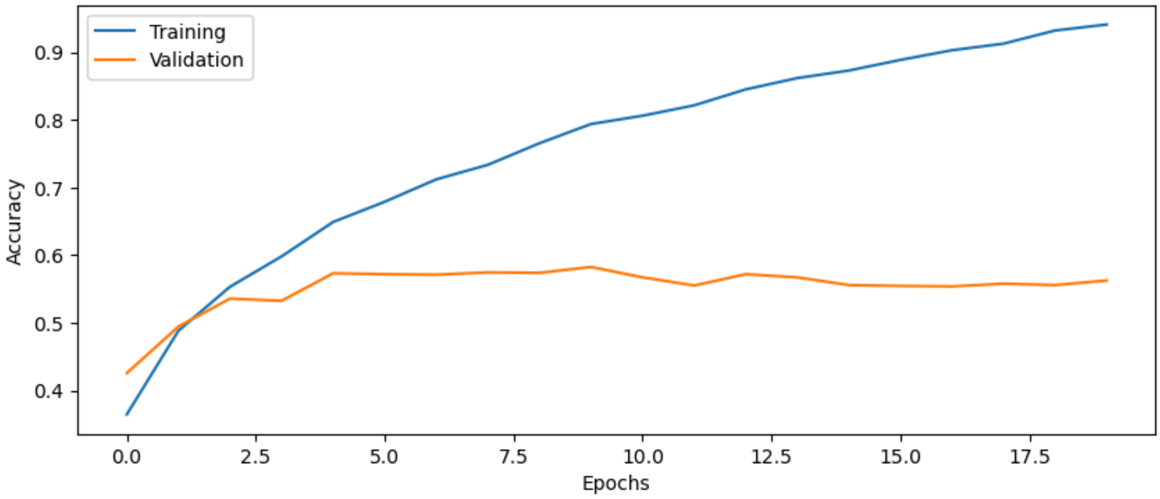
```
# --------------------------------------------
# === Your code here ========================
# --------------------------------------------

# Evaluate the trained model on test set, not used in training or validation
score = model1.evaluate(Xtest, Ytest, batch_size=batch_size)

# ============================================

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
75/75 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5518 - loss: 1.9240
Test loss: 1.9599
Test accuracy: 0.5540
```

In [15]:

```
from utilities import plot_results
# Plot the history from the training run
plot_results(history1)
```





4.3 Improving model performance

Write down the test accuracy, are you satisfied with the classifier performance (random chance is 10%)?

Questions

13. How big is the difference between training and test accuracy?

14. For the DNN elaboration we used a batch size of 10.000, why do we need to use a smaller batch size in this elaboration?

Answers

Test accuracy is 55%, quite low honestly.

13. The difference between train accuracy and test accuracy is 10%.

14. Bigger batch size would spend more time and occupy more memory on loading the batch and calculating gradients. In this case, the information in images is much more than figures, smaller batch size may release the burden of computational resource.

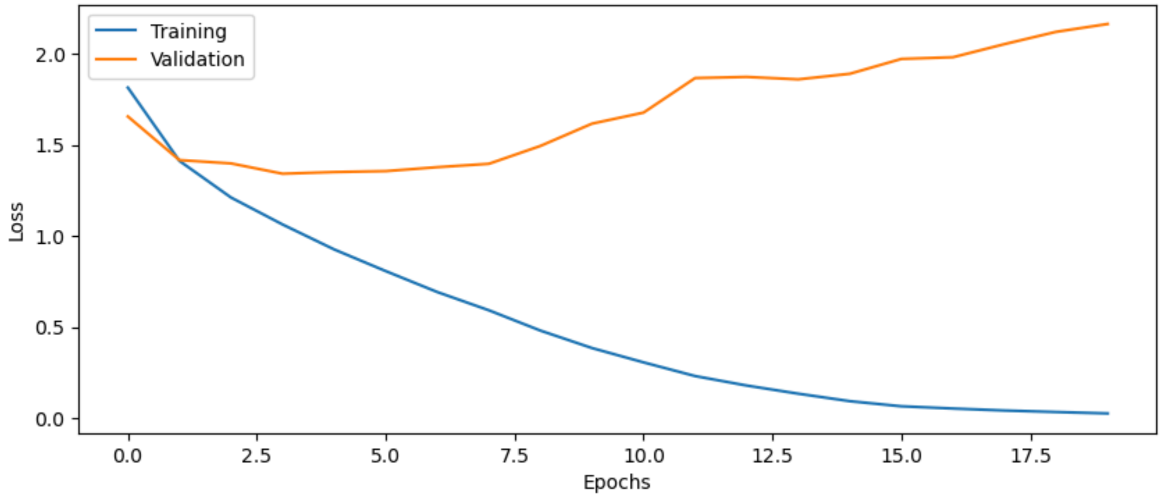
Experiment with several model configurations in the following sections.

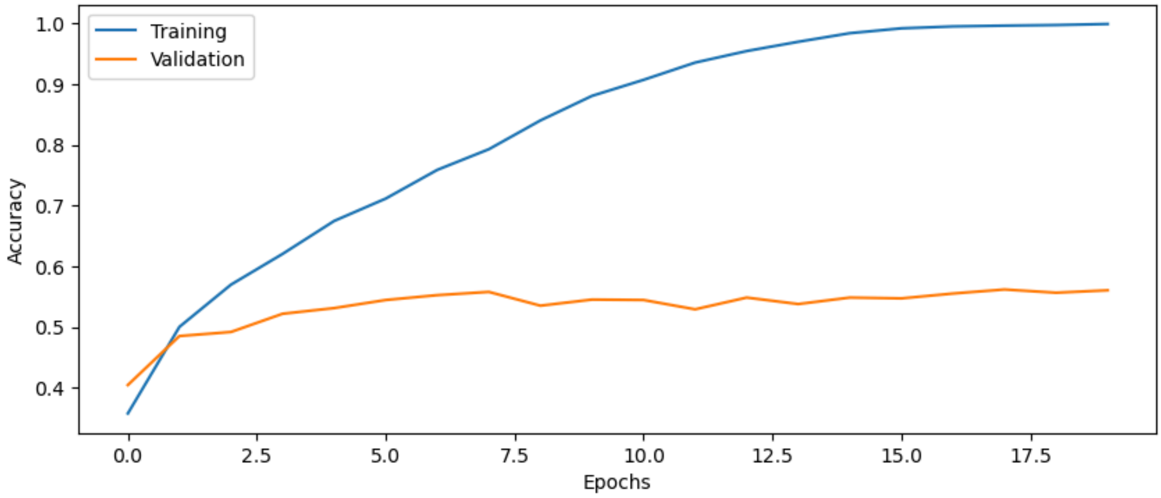
2 convolutional layers with 16 starting filters and 1 intermediate dense layer (50 nodes)

In [16]:
```
# --------------------------------------------
# === Your code here ========================
# --------------------------------------------

# Build and train model
model1 = build_CNN(input_shape=input_shape,
                   loss=CategoricalCrossentropy(),
                   learning_rate=learning_rate,
                   n_dense_layers=1)

history1 = model1.fit(Xtrain,
                      Ytrain,
                      validation_data=(Xval, Yval),
                      epochs=epochs,
                      batch_size=batch_size)

# Evaluate model on test data
score = model1.evaluate(Xtest, Ytest, batch_size=batch_size)

# ============================================

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

# Plot the history from the training run
plot_results(history1)
```

```
Epoch 1/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.3001 - loss: 2.0265 - val_
accuracy: 0.4047 - val_loss: 1.6560
Epoch 2/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.4962 - loss: 1.4203 - val_
accuracy: 0.4853 - val_loss: 1.4165
Epoch 3/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.5697 - loss: 1.2117 - val_
accuracy: 0.4920 - val_loss: 1.3983
Epoch 4/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.6227 - loss: 1.0664 - val_
accuracy: 0.5220 - val_loss: 1.3418
Epoch 5/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.6718 - loss: 0.9242 - val_
accuracy: 0.5313 - val_loss: 1.3509
Epoch 6/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.7234 - loss: 0.7720 - val_
accuracy: 0.5447 - val_loss: 1.3560
Epoch 7/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.7771 - loss: 0.6602 - val_
accuracy: 0.5527 - val_loss: 1.3776
Epoch 8/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8043 - loss: 0.5695 - val_
accuracy: 0.5580 - val_loss: 1.3961
Epoch 9/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8548 - loss: 0.4605 - val_
accuracy: 0.5353 - val_loss: 1.4942
Epoch 10/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8964 - loss: 0.3551 - val_
accuracy: 0.5453 - val_loss: 1.6166
Epoch 11/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9074 - loss: 0.3041 - val_
accuracy: 0.5447 - val_loss: 1.6773
Epoch 12/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9395 - loss: 0.2214 - val_
accuracy: 0.5293 - val_loss: 1.8669
Epoch 13/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9624 - loss: 0.1679 - val_
accuracy: 0.5487 - val_loss: 1.8732
Epoch 14/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9751 - loss: 0.1247 - val_
accuracy: 0.5380 - val_loss: 1.8598
Epoch 15/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9856 - loss: 0.0919 - val_
accuracy: 0.5487 - val_loss: 1.8905
Epoch 16/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9935 - loss: 0.0613 - val_
accuracy: 0.5473 - val_loss: 1.9720
Epoch 17/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9970 - loss: 0.0459 - val_
accuracy: 0.5553 - val_loss: 1.9812
Epoch 18/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9978 - loss: 0.0381 - val_
accuracy: 0.5620 - val_loss: 2.0533
Epoch 19/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9985 - loss: 0.0313 - val_
accuracy: 0.5567 - val_loss: 2.1207
Epoch 20/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9991 - loss: 0.0238 - val_
accuracy: 0.5607 - val_loss: 2.1637
```

```
75/75 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5528 - loss: 2.0763
Test loss: 2.1194
Test accuracy: 0.5567
```





4 convolutional layers with 16 starting filters and 1 intermediate dense layer (50 nodes)

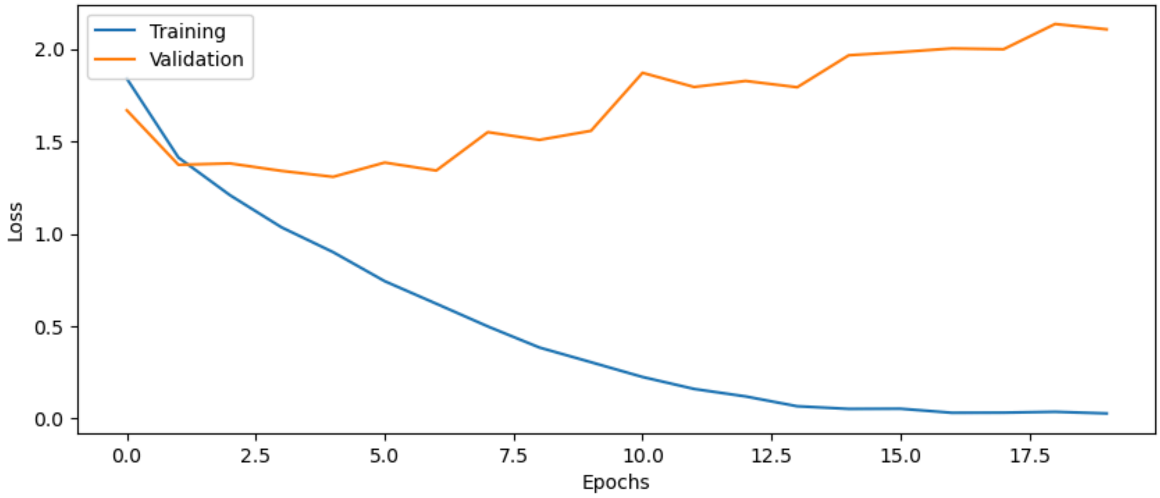
In [17]:

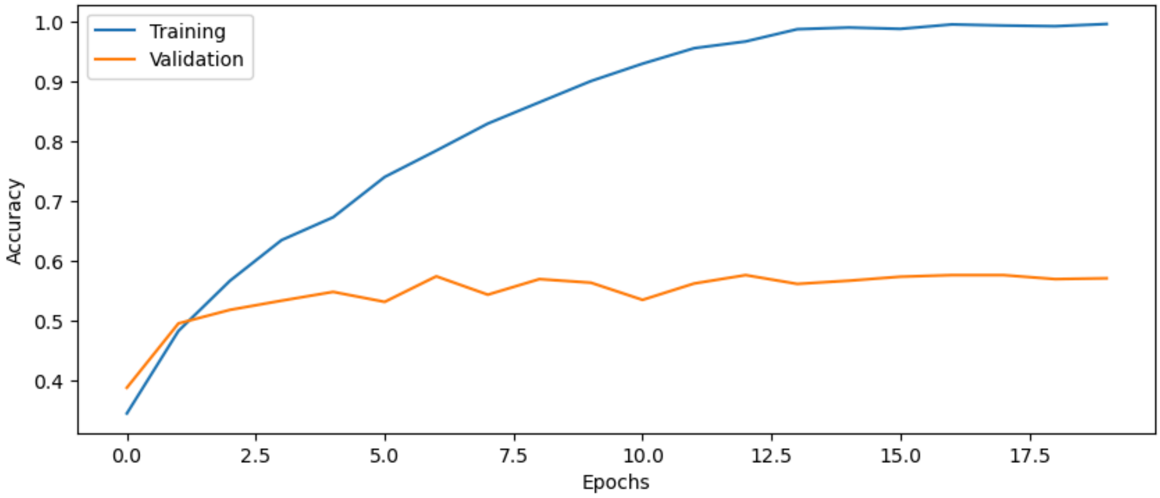
```
# -------------------------------------------
# === Your code here =======================
# -------------------------------------------

# Build and train model
model2 = build_CNN(input_shape=input_shape,
                   loss=CategoricalCrossentropy(),
                   learning_rate=learning_rate,
                   n_dense_layers=1,
                   n_conv_layers=4)

history2 = model2.fit(Xtrain,
                      Ytrain,
                      validation_data=(Xval, Yval),
                      epochs=epochs,
                      batch_size=batch_size)

# Evaluate model on test data
score = model2.evaluate(Xtest, Ytest, batch_size=batch_size)

# ===========================================
```

```
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

# Plot the history from the training run
plot_results(history2)
```

```
Epoch 1/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.2822 - loss: 2.1197 - val
_accuracy: 0.3887 - val_loss: 1.6679
Epoch 2/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.4784 - loss: 1.4214 - val
_accuracy: 0.4960 - val_loss: 1.3732
Epoch 3/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.5574 - loss: 1.2208 - val
_accuracy: 0.5187 - val_loss: 1.3806
Epoch 4/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.6421 - loss: 1.0208 - val
_accuracy: 0.5340 - val_loss: 1.3402
Epoch 5/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.6807 - loss: 0.8800 - val
_accuracy: 0.5487 - val_loss: 1.3080
Epoch 6/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.7626 - loss: 0.7019 - val
_accuracy: 0.5320 - val_loss: 1.3852
Epoch 7/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.8058 - loss: 0.5723 - val
_accuracy: 0.5747 - val_loss: 1.3421
Epoch 8/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.8400 - loss: 0.4788 - val
_accuracy: 0.5440 - val_loss: 1.5501
Epoch 9/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.8889 - loss: 0.3395 - val
_accuracy: 0.5700 - val_loss: 1.5080
Epoch 10/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9116 - loss: 0.2813 - val
_accuracy: 0.5640 - val_loss: 1.5571
Epoch 11/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9373 - loss: 0.2091 - val
_accuracy: 0.5353 - val_loss: 1.8712
Epoch 12/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9605 - loss: 0.1456 - val
_accuracy: 0.5627 - val_loss: 1.7945
Epoch 13/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9687 - loss: 0.1146 - val
_accuracy: 0.5767 - val_loss: 1.8271
Epoch 14/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9906 - loss: 0.0582 - val
_accuracy: 0.5620 - val_loss: 1.7930
Epoch 15/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9923 - loss: 0.0485 - val
_accuracy: 0.5673 - val_loss: 1.9662
Epoch 16/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9874 - loss: 0.0502 - val
_accuracy: 0.5740 - val_loss: 1.9832
Epoch 17/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9938 - loss: 0.0313 - val
_accuracy: 0.5767 - val_loss: 2.0032
Epoch 18/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9949 - loss: 0.0277 - val
_accuracy: 0.5767 - val_loss: 1.9988
Epoch 19/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9946 - loss: 0.0307 - val
_accuracy: 0.5700 - val_loss: 2.1355
Epoch 20/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9957 - loss: 0.0279 - val
_accuracy: 0.5713 - val_loss: 2.1066
```

```
75/75 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5953 - loss: 1.9853
Test loss: 2.0513
Test accuracy: 0.5807
```





4.4 Plot the CNN architecture and understand the internal model dimensions

To understand your network better, print the architecture using `model.summary()`

Questions

15. How many trainable parameters does your network have? Which part of the network contains most of the parameters?

16. What is the input to and output of a Conv2D layer? What are the dimensions of the input and output?

17. Is the batch size always the first dimension of each 4D tensor? Check the documentation for Conv2D.

18. If a convolutional layer that contains 128 filters is applied to an input with 32 channels, what is the number of channels in the output?

19. Why is the number of parameters in each Conv2D layer *not* equal to the number of filters times the number of filter coefficients per filter (plus biases)?

20. How does MaxPooling help in reducing the number of parameters to train?

Answers

15. The number of parameters is 13,018. The second convolutional layer contains the most parameters.

16. Take the first convolutional layer for example, the input is encoded image and its output is the features, the dimension of input is (32, 32, 3) and the dimension of output is (32, 32, 16).

17. Yes! Alough the parameter input size is (32, 32, 3), the batch size would be finally added in the first place.

18. The output channels would be equal to the filters number.

19. The number of parameters in convolutional layer should be (kernel size * input_channels + 1) * filter number, which represents the whole input (channel=3 in our model) should be convolved by each filter. The question ignores the input_channel.

20. The Maxpooling select the maximum value in the pool as the new feature, which equals to transform 4 value (maxpooling(2,2)) as 1.

In [18]:
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	
conv2d (Conv2D)	(None, 32, 32, 16)	
batch_normalization (BatchNormalization)	(None, 32, 32, 16)	
max_pooling2d (MaxPooling2D)	(None, 16, 16, 16)	
conv2d_1 (Conv2D)	(None, 16, 16, 32)	
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 32)	
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 32)	
flatten (Flatten)	(None, 2048)	
dense (Dense)	(None, 10)	

Total params: 25,770 (100.66 KB)

```
Trainable params: 25,674 (100.29 KB)
Non-trainable params: 96 (384.00 B)
```

4.5 Dropout regularization

Add dropout regularization between each intermediate dense layer, with dropout probability 50%.

Questions

21. How much did the test accuracy improve with dropout, compared to without dropout?

22. What other types of regularization can be applied? How can you add `L2 regularization` for the convolutional layers?

Answers

21. Nothing improvement on test accuracy, I think the model is still in a state of underfitting.

22. We can take some methods like L1-regularization, L2-regularization, early stop, and data augmentation. We can set the parameter in Conv2D: kernel_regularizer= regularizer.l2() and set the hyperparameter in parentheses.

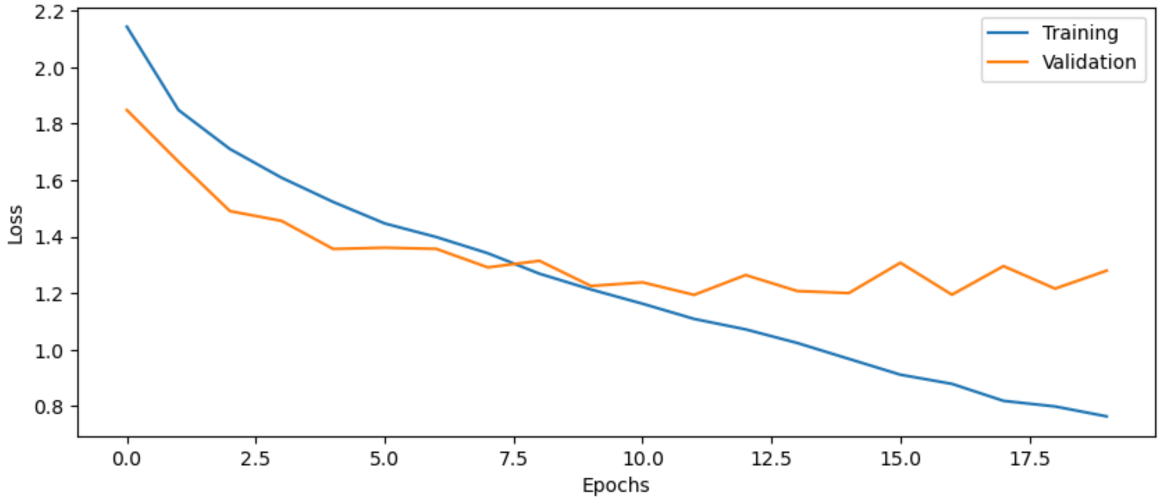
4 convolutional layers with 16 starting filters and 1 intermediate dense layer (50 nodes) with dropout

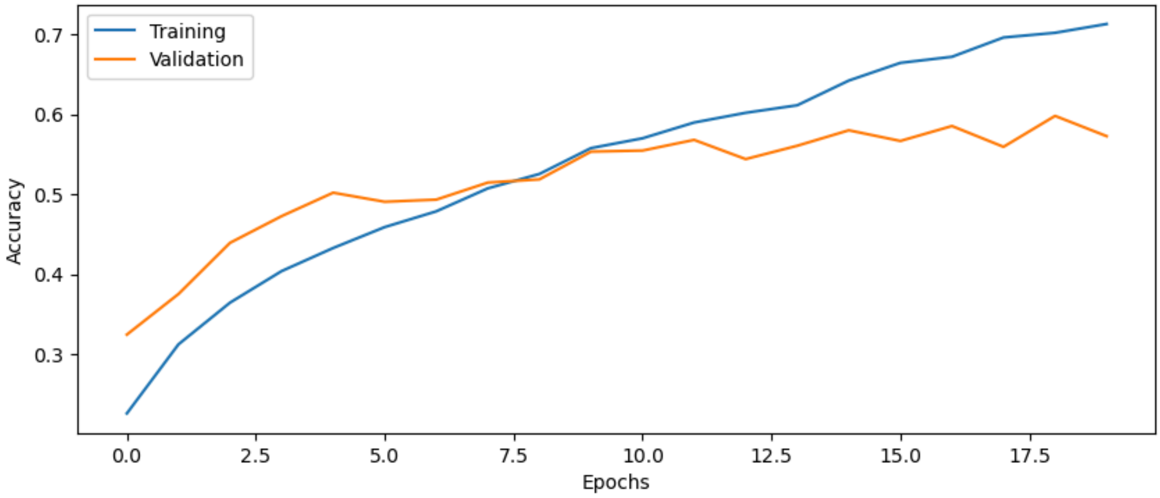
In [19]:
```python
# ---------------------------------------------
# === Your code here ========================
# ---------------------------------------------

# Setup some training parameters
batch_size = 20
epochs = 20
input_shape = input_shape
learning_rate = 0.01

# Build and train model
model3 = build_CNN(input_shape=input_shape,
                   loss=CategoricalCrossentropy(),
                   learning_rate=learning_rate,
                   n_dense_layers=1,
                   n_conv_layers=4,
                   use_dropout=True)

# Train the model using training data and validation data
history3 = model3.fit(Xtrain,
                      Ytrain,
                      validation_data=(Xval, Yval),
                      epochs=epochs,
                      batch_size=batch_size)
```

```python
# Evaluate model on test data
score = model3.evaluate(Xtest, Ytest, batch_size=batch_size)

# ============================================

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

plot_results(history3)
```

```
Epoch 1/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.1798 - loss: 2.3819 - val
_accuracy: 0.3247 - val_loss: 1.8484
Epoch 2/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.2914 - loss: 1.8946 - val
_accuracy: 0.3753 - val_loss: 1.6657
Epoch 3/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.3627 - loss: 1.7178 - val
_accuracy: 0.4393 - val_loss: 1.4907
Epoch 4/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.3901 - loss: 1.6276 - val
_accuracy: 0.4727 - val_loss: 1.4561
Epoch 5/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.4371 - loss: 1.5065 - val
_accuracy: 0.5020 - val_loss: 1.3567
Epoch 6/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.4582 - loss: 1.4272 - val
_accuracy: 0.4907 - val_loss: 1.3612
Epoch 7/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.4797 - loss: 1.3828 - val
_accuracy: 0.4933 - val_loss: 1.3570
Epoch 8/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.5066 - loss: 1.3353 - val
_accuracy: 0.5147 - val_loss: 1.2915
Epoch 9/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.5293 - loss: 1.2608 - val
_accuracy: 0.5187 - val_loss: 1.3148
Epoch 10/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.5579 - loss: 1.2170 - val
_accuracy: 0.5533 - val_loss: 1.2257
Epoch 11/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.5743 - loss: 1.1686 - val
_accuracy: 0.5547 - val_loss: 1.2385
Epoch 12/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.5993 - loss: 1.0835 - val
_accuracy: 0.5680 - val_loss: 1.1942
Epoch 13/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 12ms/step - accuracy: 0.6026 - loss: 1.0492 - val
_accuracy: 0.5440 - val_loss: 1.2643
Epoch 14/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.6162 - loss: 0.9996 - val
_accuracy: 0.5607 - val_loss: 1.2078
Epoch 15/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.6472 - loss: 0.9393 - val
_accuracy: 0.5800 - val_loss: 1.2005
Epoch 16/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.6640 - loss: 0.9032 - val
_accuracy: 0.5667 - val_loss: 1.3079
Epoch 17/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.6669 - loss: 0.8698 - val
_accuracy: 0.5853 - val_loss: 1.1949
Epoch 18/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.7055 - loss: 0.8022 - val
_accuracy: 0.5593 - val_loss: 1.2960
Epoch 19/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.7132 - loss: 0.7813 - val
_accuracy: 0.5980 - val_loss: 1.2163
Epoch 20/20
350/350 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.7250 - loss: 0.7427 - val
_accuracy: 0.5727 - val_loss: 1.2800
```

```
75/75 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5851 - loss: 1.2145
Test loss: 1.2433
Test accuracy: 0.5800
```





4.6 Tweaking model performance

You have now seen the basic building blocks of a 2D CNN. To further improve performance involves changing the number of convolutional layers, the number of filters per layer, the number of intermediate dense layers, the number of nodes in the intermediate dense layers, batch size, learning rate, number of epochs, etc. Spend some time (30 - 90 minutes) testing different settings.

Questions

23. How high test accuracy can you obtain? What is your best configuration?

Answers

23. Unfortunately, 55% on test data, I used l2-regularization, dropout and adam optimizer.

Your best config

In [20]:

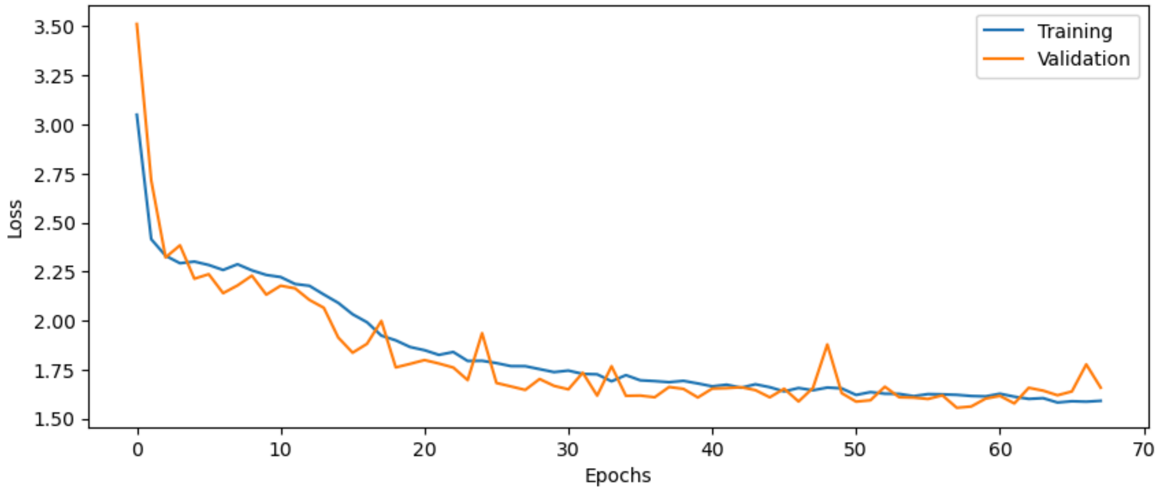
```
# --------------------------------------------
# === Your code here =========================
# --------------------------------------------
from tensorflow.keras.callbacks import EarlyStopping
# Setup some training parameters
batch_size = 32
epochs = 100
input_shape = input_shape
learning_rate = 0.005

# Build and train model. Here experiment with several model architecture configu
model4 = build_CNN(input_shape=input_shape,
                   loss=CategoricalCrossentropy(),
                   learning_rate=learning_rate,
                   n_dense_layers=1,
                   n_conv_layers=3,
                   use_dropout=True,
                   l2_regularization=True,
                   optimizer='adam')

# early stop
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=10,
                               restore_best_weights=True)

history4 = model4.fit(Xtrain,
                      Ytrain,
                      validation_data=(Xval, Yval),
                      epochs=epochs,
                      batch_size=batch_size,
                      callbacks=[early_stopping])

# Evaluate model on test data
score = model4.evaluate(Xtest, Ytest, batch_size=batch_size)


# ============================================

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

# Plot the history from the training run
plot_results(history4)
```

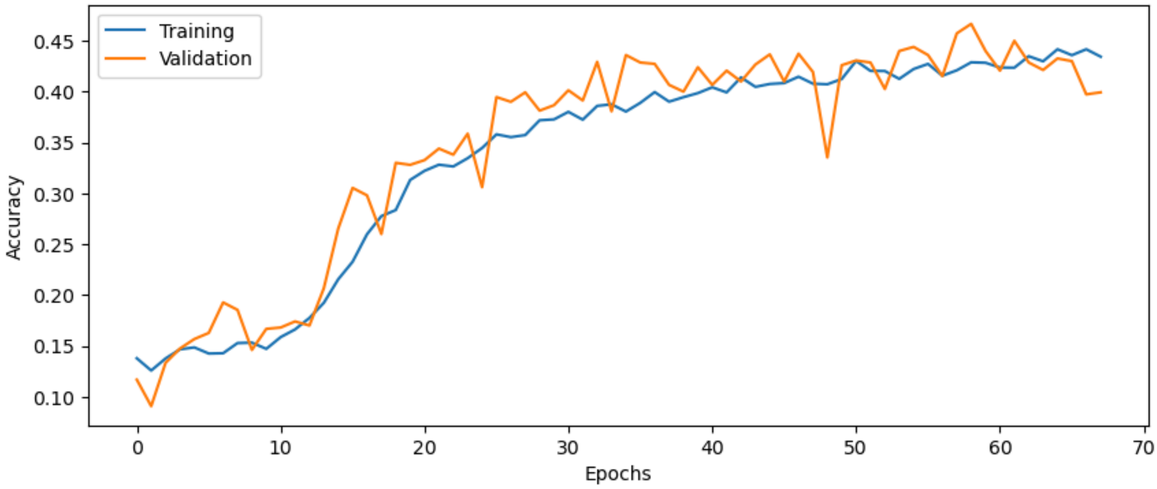
```
Epoch 1/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.1444 - loss: 3.7619 - val
_accuracy: 0.1167 - val_loss: 3.5115
Epoch 2/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1317 - loss: 2.4397 - val
_accuracy: 0.0907 - val_loss: 2.7148
Epoch 3/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1357 - loss: 2.3328 - val
_accuracy: 0.1333 - val_loss: 2.3221
Epoch 4/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1460 - loss: 2.2843 - val
_accuracy: 0.1473 - val_loss: 2.3838
Epoch 5/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1555 - loss: 2.3097 - val
_accuracy: 0.1567 - val_loss: 2.2135
Epoch 6/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - accuracy: 0.1444 - loss: 2.2896 - val
_accuracy: 0.1627 - val_loss: 2.2366
Epoch 7/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1444 - loss: 2.2495 - val
_accuracy: 0.1927 - val_loss: 2.1398
Epoch 8/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - accuracy: 0.1570 - loss: 2.2850 - val
_accuracy: 0.1853 - val_loss: 2.1792
Epoch 9/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1508 - loss: 2.2949 - val
_accuracy: 0.1460 - val_loss: 2.2292
Epoch 10/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1425 - loss: 2.2546 - val
_accuracy: 0.1667 - val_loss: 2.1324
Epoch 11/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1546 - loss: 2.2239 - val
_accuracy: 0.1680 - val_loss: 2.1780
Epoch 12/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1638 - loss: 2.1931 - val
_accuracy: 0.1740 - val_loss: 2.1644
Epoch 13/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1867 - loss: 2.1764 - val
_accuracy: 0.1700 - val_loss: 2.1054
Epoch 14/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.1776 - loss: 2.1505 - val
_accuracy: 0.2067 - val_loss: 2.0652
Epoch 15/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.2039 - loss: 2.1084 - val
_accuracy: 0.2653 - val_loss: 1.9142
Epoch 16/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.2266 - loss: 2.0413 - val
_accuracy: 0.3053 - val_loss: 1.8365
Epoch 17/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.2548 - loss: 2.0057 - val
_accuracy: 0.2980 - val_loss: 1.8817
Epoch 18/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.2615 - loss: 1.9277 - val
_accuracy: 0.2600 - val_loss: 1.9984
Epoch 19/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.2743 - loss: 1.9133 - val
_accuracy: 0.3300 - val_loss: 1.7616
Epoch 20/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3056 - loss: 1.8909 - val
_accuracy: 0.3280 - val_loss: 1.7797
```

```
Epoch 21/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3301 - loss: 1.8188 - val
_accuracy: 0.3327 - val_loss: 1.7990
Epoch 22/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3377 - loss: 1.8107 - val
_accuracy: 0.3440 - val_loss: 1.7814
Epoch 23/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3274 - loss: 1.8402 - val
_accuracy: 0.3380 - val_loss: 1.7609
Epoch 24/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3314 - loss: 1.7865 - val
_accuracy: 0.3587 - val_loss: 1.6968
Epoch 25/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3475 - loss: 1.7772 - val
_accuracy: 0.3060 - val_loss: 1.9362
Epoch 26/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3593 - loss: 1.8001 - val
_accuracy: 0.3947 - val_loss: 1.6822
Epoch 27/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3679 - loss: 1.7335 - val
_accuracy: 0.3900 - val_loss: 1.6647
Epoch 28/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3622 - loss: 1.7574 - val
_accuracy: 0.3993 - val_loss: 1.6469
Epoch 29/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3781 - loss: 1.7372 - val
_accuracy: 0.3813 - val_loss: 1.7024
Epoch 30/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3843 - loss: 1.7114 - val
_accuracy: 0.3867 - val_loss: 1.6676
Epoch 31/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3886 - loss: 1.7466 - val
_accuracy: 0.4013 - val_loss: 1.6498
Epoch 32/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3660 - loss: 1.7178 - val
_accuracy: 0.3913 - val_loss: 1.7345
Epoch 33/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3872 - loss: 1.7256 - val
_accuracy: 0.4293 - val_loss: 1.6179
Epoch 34/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3910 - loss: 1.6690 - val
_accuracy: 0.3807 - val_loss: 1.7676
Epoch 35/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3846 - loss: 1.7057 - val
_accuracy: 0.4360 - val_loss: 1.6167
Epoch 36/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3862 - loss: 1.6988 - val
_accuracy: 0.4287 - val_loss: 1.6180
Epoch 37/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4028 - loss: 1.6811 - val
_accuracy: 0.4273 - val_loss: 1.6095
Epoch 38/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4058 - loss: 1.6847 - val
_accuracy: 0.4067 - val_loss: 1.6618
Epoch 39/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4003 - loss: 1.6540 - val
_accuracy: 0.4000 - val_loss: 1.6520
Epoch 40/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3994 - loss: 1.6776 - val
_accuracy: 0.4240 - val_loss: 1.6084
```

```
Epoch 41/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4104 - loss: 1.6658 - val
_accuracy: 0.4067 - val_loss: 1.6536
Epoch 42/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4026 - loss: 1.6689 - val
_accuracy: 0.4207 - val_loss: 1.6559
Epoch 43/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.3989 - loss: 1.6744 - val
_accuracy: 0.4100 - val_loss: 1.6606
Epoch 44/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4045 - loss: 1.6770 - val
_accuracy: 0.4267 - val_loss: 1.6457
Epoch 45/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4100 - loss: 1.6478 - val
_accuracy: 0.4367 - val_loss: 1.6092
Epoch 46/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4130 - loss: 1.6271 - val
_accuracy: 0.4100 - val_loss: 1.6528
Epoch 47/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4126 - loss: 1.6490 - val
_accuracy: 0.4373 - val_loss: 1.5880
Epoch 48/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4065 - loss: 1.6496 - val
_accuracy: 0.4193 - val_loss: 1.6568
Epoch 49/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4155 - loss: 1.6255 - val
_accuracy: 0.3353 - val_loss: 1.8785
Epoch 50/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4013 - loss: 1.6656 - val
_accuracy: 0.4260 - val_loss: 1.6304
Epoch 51/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4467 - loss: 1.5950 - val
_accuracy: 0.4307 - val_loss: 1.5872
Epoch 52/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4169 - loss: 1.6281 - val
_accuracy: 0.4287 - val_loss: 1.5942
Epoch 53/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4219 - loss: 1.6108 - val
_accuracy: 0.4027 - val_loss: 1.6632
Epoch 54/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4202 - loss: 1.6007 - val
_accuracy: 0.4400 - val_loss: 1.6094
Epoch 55/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4277 - loss: 1.5808 - val
_accuracy: 0.4440 - val_loss: 1.6076
Epoch 56/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4334 - loss: 1.6100 - val
_accuracy: 0.4360 - val_loss: 1.6008
Epoch 57/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4194 - loss: 1.6257 - val
_accuracy: 0.4153 - val_loss: 1.6191
Epoch 58/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4250 - loss: 1.6126 - val
_accuracy: 0.4573 - val_loss: 1.5558
Epoch 59/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4236 - loss: 1.6118 - val
_accuracy: 0.4667 - val_loss: 1.5623
Epoch 60/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4155 - loss: 1.6382 - val
_accuracy: 0.4400 - val_loss: 1.6025
```

```
Epoch 61/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4291 - loss: 1.6171 - val
_accuracy: 0.4207 - val_loss: 1.6164
Epoch 62/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4212 - loss: 1.6085 - val
_accuracy: 0.4500 - val_loss: 1.5780
Epoch 63/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4441 - loss: 1.5804 - val
_accuracy: 0.4287 - val_loss: 1.6576
Epoch 64/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4232 - loss: 1.6137 - val
_accuracy: 0.4213 - val_loss: 1.6432
Epoch 65/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4440 - loss: 1.5636 - val
_accuracy: 0.4327 - val_loss: 1.6196
Epoch 66/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4393 - loss: 1.5817 - val
_accuracy: 0.4300 - val_loss: 1.6390
Epoch 67/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4466 - loss: 1.5751 - val
_accuracy: 0.3973 - val_loss: 1.7769
Epoch 68/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.4216 - loss: 1.6003 - val
_accuracy: 0.3993 - val_loss: 1.6585
47/47 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4495 - loss: 1.5817
Test loss: 1.5646
Test accuracy: 0.4480
```





Part 5: Model generalization

How high is the test accuracy if we rotate the test images? In other words, how good is the CNN at generalizing to rotated images?

Rotate each test image 90 degrees, the cells are already finished.

Questions

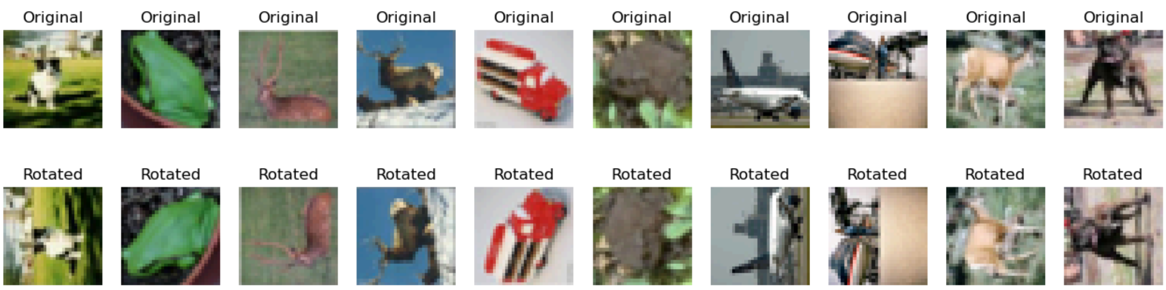
24. What is the test accuracy for rotated test images, compared to test images without rotation? Explain the difference in accuracy.

Answers

Test accuracy is extremely low (0.20), performing substantially worse than a random guess. Since CNN learns filters that detect features like edges and shapes, when images are rotated it creates a domain shift - it didn't learn how to recognize features in this orientation.

In [21]:

```
from utilities import myrotate
# Visualize some rotated images
# Rotate the test images 90 degrees
Xtest_rotated = myrotate(Xtest)

# Look at some rotated images
plt.figure(figsize=(16,4))
for i in range(10):
    idx = np.random.randint(500)

    plt.subplot(2,10,i+1)
    plt.imshow(Xtest[idx]/2+0.5)
    plt.title("Original")
    plt.axis('off')

    plt.subplot(2,10,i+11)
    plt.imshow(Xtest_rotated[idx]/2+0.5)
    plt.title("Rotated")
    plt.axis('off')
plt.show()
```



In [22]:

```
# Evaluate the trained model on rotated test set
score = model4.evaluate(Xtest_rotated, Ytest, verbose=0)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
Test loss: 2.7262
Test accuracy: 0.1867
```

5.1 Augmentation using Keras `ImageDataGenerator`

We can increase the number of training images through data augmentation (we now ignore that CIFAR10 actually has 60 000 training images). Image augmentation is about creating similar images, by performing operations such as rotation, scaling, elastic deformations and flipping of existing images. This will prevent overfitting, especially if all the training images are in a certain orientation.

We will perform the augmentation on the fly, using a built-in function in Keras, called `ImageDataGenerator`. In particular, we will use the `flow()` functionality (see the documentation for more details).

Make sure to use different subsets for training and validation when you calling `flow()` on the training data generator in `model.fit()`, otherwise you will validate on the same data.

In [23]:

```python
# Get all 60 000 training images again. ImageDataGenerator manages validation da

# re-load the CIFAR10 train and test data
(X, Y), (Xtest, Ytest) = cifar10.load_data()

# Reduce the number of images for training/validation and testing to 10000 and 2
# to reduce processing time for this elaboration.
X = X[0:10000]
Y = Y[0:10000]

Xtest = Xtest[0:2000]
Ytest = Ytest[0:2000]

# Change data type and rescale range
X = X.astype('float32')
Xtest = Xtest.astype('float32')

X = X / 127.5 - 1
Xtest = Xtest / 127.5 - 1


# Convert labels to hot encoding
Y = to_categorical(Y, 10)
Ytest = to_categorical(Ytest, 10)

print("Training/validation images have size {} and labels have size {} ".format(
print("Test images have size {} and labels have size {} \n ".format(Xtest.shape,
```

```
Training/validation images have size (10000, 32, 32, 3) and labels have size (100
00, 10)
Test images have size (2000, 32, 32, 3) and labels have size (2000, 10)
```

In [25]:

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

```
# ---------------------------------------------
# === Your code here ==========================
# ---------------------------------------------

# Use a rotation range of 30 degrees, horizontal and vertical flipping
# Set up image data generator
image_dataset = ImageDataGenerator(rotation_range=30, horizontal_flip=True, vert

train_flow = image_dataset.flow(X,Y, batch_size=1)

# =============================================
```

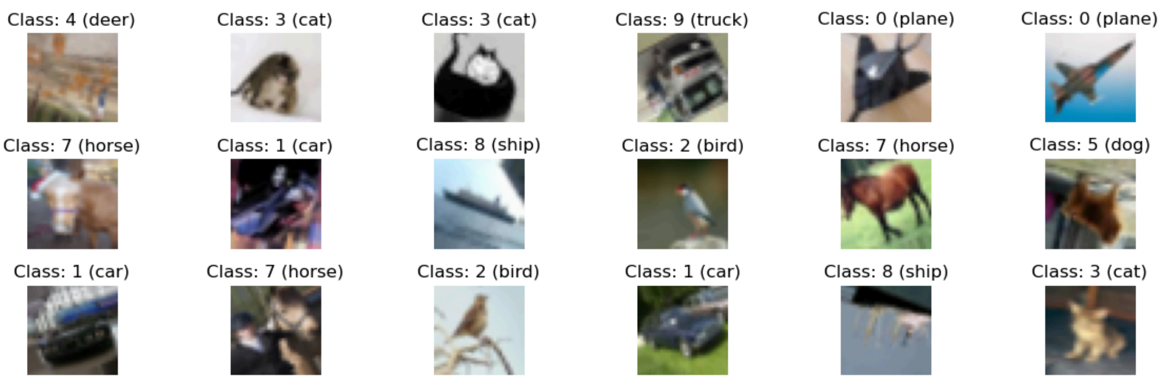
Questions

25. How would you change the code for the image generator if you cannot fit all training images in CPU memory? What is the disadvantage of doing that change?

Answers

One could use flow_from_directory("path/to/train/data", target_size=(nn,nn), batch_size=nn, class_mode="categorical") - but disk is slower than memory, so might result in slower training.

In [27]:
```
# Plot some augmented images

plt.figure(figsize=(12,4))
for i in range(18):
    (im, label) = next(train_flow)
    im = (im[0] + 1) * 127.5
    im = im.astype('int')
    label = np.flatnonzero(label)[0]

    plt.subplot(3,6,i+1)
    plt.tight_layout()
    plt.imshow(im)
    plt.title("Class: {} ({})".format(label, classes[label]))
    plt.axis('off')
plt.show()
```



5.2 Train the CNN with images from the generator

Check the documentation for the `model.fit` method how to use it with a generator instead of a fix dataset (numpy arrays).

To make the comparison fair to training without augmentation

- `steps_per_epoch` should be set to: `len(Xtrain)/batch_size`
- `validation_steps` should be set to: `len(Xval)/batch_size`

This is required since with a generator, the fit function will not know how many examples your original dataset has.

Questions

26. How quickly is the training accuracy increasing compared to without augmentation? Explain why there is a difference compared to without augmentation. We are here talking about the number of training epochs required to reach a certain accuracy, and not the training time in seconds. What parameter is necessary to change to perform more training?

27. What other types of image augmentation can be applied, compared to what we use here?

Answers

26. Training accuracy increases slower in comparison, because it takes more epochs for the model to reach a certain training accuracy - each epoch sees different versions of the same pictures. It is a sort of regularization.

27. One can zoom, shear, adjust brightness, contrast, shift channels, width and height, for instance.

In [28]:
```
# --------------------------------------------
# === Your code here ========================
# --------------------------------------------

# Setup training parameters
batch_size = 64
epochs = 500
input_shape = Xtest[1].shape

# Build model (your best config)
model6 = build_CNN(
    input_shape=input_shape,
    loss=CategoricalCrossentropy(),
    learning_rate=0.0005,
    n_dense_layers=1,
    n_conv_layers=3,
    use_dropout=True,
    l2_regularization=True,
    optimizer="adam"
)

# A workaround converting NumpyArrayIterator to a standard generator yielding ba
def wrap_iterator(numpy_iterator):
```

```
    while True:
        X_batch, Y_batch = next(numpy_iterator)
        yield X_batch, Y_batch

X_train = X[:8000]
Y_train = Y[:8000]
X_val   = X[8000:]
Y_val   = Y[8000:]

# Set up training and validation dataset flows from image_dataset
# flow() for training data
train_flow = image_dataset.flow(X_train, Y_train, batch_size=batch_size)

# flow() for validation data
val_flow = image_dataset.flow(X_val, Y_val, batch_size=batch_size)

# Wrapping iterator - otherwise there's an error
train_gen = wrap_iterator(train_flow)
val_gen = wrap_iterator(val_flow)

# Train the model using on the fly augmentation
history6 = model6.fit(
    train_gen,
    steps_per_epoch=len(X_train) // batch_size,
    validation_data=val_gen,
    validation_steps=len(X_val) // batch_size,
    epochs=epochs
)

# ============================================
```

```
Epoch 1/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 10s 62ms/step - accuracy: 0.1546 - loss: 4.5972 - va
l_accuracy: 0.1084 - val_loss: 4.6393
Epoch 2/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.2459 - loss: 3.6095 - val
_accuracy: 0.1119 - val_loss: 5.0516
Epoch 3/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.2723 - loss: 3.1266 - val
_accuracy: 0.1542 - val_loss: 3.8031
Epoch 4/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.3006 - loss: 2.7838 - val
_accuracy: 0.2794 - val_loss: 2.7020
Epoch 5/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 64ms/step - accuracy: 0.3189 - loss: 2.5322 - val
_accuracy: 0.3528 - val_loss: 2.3698
Epoch 6/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.3306 - loss: 2.3346 - val
_accuracy: 0.3709 - val_loss: 2.1943
Epoch 7/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.3413 - loss: 2.2168 - val
_accuracy: 0.4044 - val_loss: 2.0385
Epoch 8/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.3532 - loss: 2.1101 - val
_accuracy: 0.4360 - val_loss: 1.9314
Epoch 9/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.3586 - loss: 2.0298 - val
_accuracy: 0.4106 - val_loss: 1.9239
Epoch 10/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.3659 - loss: 1.9801 - val
_accuracy: 0.4380 - val_loss: 1.8513
Epoch 11/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.3829 - loss: 1.9159 - val
_accuracy: 0.4143 - val_loss: 1.8934
Epoch 12/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.3878 - loss: 1.8920 - val
_accuracy: 0.4318 - val_loss: 1.7909
Epoch 13/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.3992 - loss: 1.8208 - val
_accuracy: 0.4623 - val_loss: 1.7292
Epoch 14/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.4009 - loss: 1.8174 - val
_accuracy: 0.4623 - val_loss: 1.7004
Epoch 15/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.4121 - loss: 1.7769 - val
_accuracy: 0.4762 - val_loss: 1.6712
Epoch 16/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.4306 - loss: 1.7725 - val
_accuracy: 0.4700 - val_loss: 1.6597
Epoch 17/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.4243 - loss: 1.7379 - val
_accuracy: 0.4938 - val_loss: 1.6837
Epoch 18/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 57ms/step - accuracy: 0.4421 - loss: 1.7002 - val
_accuracy: 0.4876 - val_loss: 1.5699
Epoch 19/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.4262 - loss: 1.6922 - val
_accuracy: 0.5150 - val_loss: 1.5529
Epoch 20/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.4428 - loss: 1.6727 - val
_accuracy: 0.4675 - val_loss: 1.6168
```

```
Epoch 21/500
125/125  7s 58ms/step - accuracy: 0.4488 - loss: 1.6655 - val
_accuracy: 0.5150 - val_loss: 1.5131
Epoch 22/500
125/125  7s 58ms/step - accuracy: 0.4712 - loss: 1.6264 - val
_accuracy: 0.5021 - val_loss: 1.5208
Epoch 23/500
125/125  7s 59ms/step - accuracy: 0.4512 - loss: 1.6214 - val
_accuracy: 0.4762 - val_loss: 1.6055
Epoch 24/500
125/125  7s 59ms/step - accuracy: 0.4620 - loss: 1.6325 - val
_accuracy: 0.5083 - val_loss: 1.5091
Epoch 25/500
125/125  7s 58ms/step - accuracy: 0.4603 - loss: 1.6294 - val
_accuracy: 0.5083 - val_loss: 1.5347
Epoch 26/500
125/125  7s 58ms/step - accuracy: 0.4644 - loss: 1.5896 - val
_accuracy: 0.4948 - val_loss: 1.5329
Epoch 27/500
125/125  7s 58ms/step - accuracy: 0.4654 - loss: 1.5968 - val
_accuracy: 0.5325 - val_loss: 1.4922
Epoch 28/500
125/125  7s 58ms/step - accuracy: 0.4825 - loss: 1.5562 - val
_accuracy: 0.5088 - val_loss: 1.5119
Epoch 29/500
125/125  7s 58ms/step - accuracy: 0.4875 - loss: 1.5828 - val
_accuracy: 0.5083 - val_loss: 1.4815
Epoch 30/500
125/125  7s 58ms/step - accuracy: 0.4827 - loss: 1.5755 - val
_accuracy: 0.5300 - val_loss: 1.4624
Epoch 31/500
125/125  7s 58ms/step - accuracy: 0.4950 - loss: 1.5372 - val
_accuracy: 0.5300 - val_loss: 1.4797
Epoch 32/500
125/125  7s 58ms/step - accuracy: 0.4962 - loss: 1.5380 - val
_accuracy: 0.5310 - val_loss: 1.4498
Epoch 33/500
125/125  7s 59ms/step - accuracy: 0.4907 - loss: 1.5594 - val
_accuracy: 0.5262 - val_loss: 1.4652
Epoch 34/500
125/125  7s 58ms/step - accuracy: 0.5049 - loss: 1.5422 - val
_accuracy: 0.5212 - val_loss: 1.4671
Epoch 35/500
125/125  7s 58ms/step - accuracy: 0.5088 - loss: 1.5457 - val
_accuracy: 0.5403 - val_loss: 1.4669
Epoch 36/500
125/125  7s 58ms/step - accuracy: 0.5095 - loss: 1.5147 - val
_accuracy: 0.5145 - val_loss: 1.5077
Epoch 37/500
125/125  7s 58ms/step - accuracy: 0.5076 - loss: 1.5165 - val
_accuracy: 0.5444 - val_loss: 1.4424
Epoch 38/500
125/125  7s 58ms/step - accuracy: 0.4996 - loss: 1.5158 - val
_accuracy: 0.5088 - val_loss: 1.4937
Epoch 39/500
125/125  7s 58ms/step - accuracy: 0.5064 - loss: 1.4844 - val
_accuracy: 0.5294 - val_loss: 1.4711
Epoch 40/500
125/125  9s 70ms/step - accuracy: 0.5134 - loss: 1.4548 - val
_accuracy: 0.5305 - val_loss: 1.4592
```

```
Epoch 41/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.4985 - loss: 1.5131 - val
_accuracy: 0.5150 - val_loss: 1.4769
Epoch 42/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.5065 - loss: 1.4989 - val
_accuracy: 0.5620 - val_loss: 1.3915
Epoch 43/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.5086 - loss: 1.4832 - val
_accuracy: 0.5491 - val_loss: 1.4153
Epoch 44/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.5132 - loss: 1.5006 - val
_accuracy: 0.5212 - val_loss: 1.4777
Epoch 45/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5297 - loss: 1.4665 - val
_accuracy: 0.5470 - val_loss: 1.4072
Epoch 46/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.5093 - loss: 1.4761 - val
_accuracy: 0.5346 - val_loss: 1.4551
Epoch 47/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.5328 - loss: 1.4456 - val
_accuracy: 0.5434 - val_loss: 1.4271
Epoch 48/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5299 - loss: 1.4550 - val
_accuracy: 0.5377 - val_loss: 1.4669
Epoch 49/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5321 - loss: 1.4530 - val
_accuracy: 0.5429 - val_loss: 1.4344
Epoch 50/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.5287 - loss: 1.4348 - val
_accuracy: 0.5320 - val_loss: 1.4469
Epoch 51/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5151 - loss: 1.4916 - val
_accuracy: 0.5522 - val_loss: 1.4720
Epoch 52/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 58ms/step - accuracy: 0.5298 - loss: 1.4526 - val
_accuracy: 0.5103 - val_loss: 1.5340
Epoch 53/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5458 - loss: 1.4527 - val
_accuracy: 0.5718 - val_loss: 1.3357
Epoch 54/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5372 - loss: 1.4237 - val
_accuracy: 0.5671 - val_loss: 1.4062
Epoch 55/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5446 - loss: 1.4385 - val
_accuracy: 0.5646 - val_loss: 1.3757
Epoch 56/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5375 - loss: 1.4283 - val
_accuracy: 0.5527 - val_loss: 1.4195
Epoch 57/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5422 - loss: 1.4226 - val
_accuracy: 0.5620 - val_loss: 1.3961
Epoch 58/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5385 - loss: 1.4204 - val
_accuracy: 0.5615 - val_loss: 1.4009
Epoch 59/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5270 - loss: 1.4355 - val
_accuracy: 0.5465 - val_loss: 1.4043
Epoch 60/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5415 - loss: 1.4210 - val
_accuracy: 0.5558 - val_loss: 1.3811
```

```
Epoch 61/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5535 - loss: 1.4041 - val
_accuracy: 0.5646 - val_loss: 1.3697
Epoch 62/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5473 - loss: 1.4193 - val
_accuracy: 0.5728 - val_loss: 1.3600
Epoch 63/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5613 - loss: 1.3750 - val
_accuracy: 0.5517 - val_loss: 1.4019
Epoch 64/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5354 - loss: 1.4506 - val
_accuracy: 0.5630 - val_loss: 1.3901
Epoch 65/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5507 - loss: 1.4051 - val
_accuracy: 0.5600 - val_loss: 1.3742
Epoch 66/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5538 - loss: 1.3973 - val
_accuracy: 0.5701 - val_loss: 1.3644
Epoch 67/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5615 - loss: 1.3886 - val
_accuracy: 0.5746 - val_loss: 1.3759
Epoch 68/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5623 - loss: 1.3683 - val
_accuracy: 0.5382 - val_loss: 1.4583
Epoch 69/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5601 - loss: 1.3674 - val
_accuracy: 0.5542 - val_loss: 1.3674
Epoch 70/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5470 - loss: 1.4170 - val
_accuracy: 0.5646 - val_loss: 1.3767
Epoch 71/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5741 - loss: 1.3591 - val
_accuracy: 0.5687 - val_loss: 1.4160
Epoch 72/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5445 - loss: 1.4064 - val
_accuracy: 0.5548 - val_loss: 1.4329
Epoch 73/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5547 - loss: 1.4170 - val
_accuracy: 0.5677 - val_loss: 1.3442
Epoch 74/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5638 - loss: 1.3827 - val
_accuracy: 0.5785 - val_loss: 1.3602
Epoch 75/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5408 - loss: 1.4141 - val
_accuracy: 0.5806 - val_loss: 1.3496
Epoch 76/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5502 - loss: 1.3848 - val
_accuracy: 0.5976 - val_loss: 1.3087
Epoch 77/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5687 - loss: 1.3776 - val
_accuracy: 0.5589 - val_loss: 1.3658
Epoch 78/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5616 - loss: 1.3741 - val
_accuracy: 0.5790 - val_loss: 1.3512
Epoch 79/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5730 - loss: 1.3653 - val
_accuracy: 0.5630 - val_loss: 1.3286
Epoch 80/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5686 - loss: 1.3563 - val
_accuracy: 0.5832 - val_loss: 1.3322
```

```
Epoch 81/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5769 - loss: 1.3635 - val
_accuracy: 0.5811 - val_loss: 1.3417
Epoch 82/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5698 - loss: 1.3524 - val
_accuracy: 0.5677 - val_loss: 1.3714
Epoch 83/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5707 - loss: 1.3833 - val
_accuracy: 0.5692 - val_loss: 1.3221
Epoch 84/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5821 - loss: 1.3632 - val
_accuracy: 0.6095 - val_loss: 1.2796
Epoch 85/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5643 - loss: 1.3751 - val
_accuracy: 0.5764 - val_loss: 1.3469
Epoch 86/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5823 - loss: 1.3298 - val
_accuracy: 0.6033 - val_loss: 1.3052
Epoch 87/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5717 - loss: 1.3730 - val
_accuracy: 0.5739 - val_loss: 1.3281
Epoch 88/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5655 - loss: 1.3765 - val
_accuracy: 0.5811 - val_loss: 1.3398
Epoch 89/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5826 - loss: 1.3388 - val
_accuracy: 0.5966 - val_loss: 1.2832
Epoch 90/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5672 - loss: 1.3587 - val
_accuracy: 0.5718 - val_loss: 1.3576
Epoch 91/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5683 - loss: 1.3675 - val
_accuracy: 0.5857 - val_loss: 1.3703
Epoch 92/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5884 - loss: 1.3229 - val
_accuracy: 0.5749 - val_loss: 1.3313
Epoch 93/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5720 - loss: 1.3556 - val
_accuracy: 0.5956 - val_loss: 1.2897
Epoch 94/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5830 - loss: 1.3288 - val
_accuracy: 0.5816 - val_loss: 1.3387
Epoch 95/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5754 - loss: 1.3463 - val
_accuracy: 0.5842 - val_loss: 1.3199
Epoch 96/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.5779 - loss: 1.3405 - val
_accuracy: 0.6095 - val_loss: 1.3051
Epoch 97/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5856 - loss: 1.3314 - val
_accuracy: 0.5796 - val_loss: 1.3402
Epoch 98/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.5733 - loss: 1.3413 - val
_accuracy: 0.5751 - val_loss: 1.3591
Epoch 99/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5889 - loss: 1.3393 - val
_accuracy: 0.5983 - val_loss: 1.2862
Epoch 100/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5773 - loss: 1.3606 - val
_accuracy: 0.5847 - val_loss: 1.3255
```

```
Epoch 101/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5788 - loss: 1.3373 - val
_accuracy: 0.5837 - val_loss: 1.3231
Epoch 102/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.5792 - loss: 1.3283 - val
_accuracy: 0.5888 - val_loss: 1.3060
Epoch 103/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5846 - loss: 1.3196 - val
_accuracy: 0.5899 - val_loss: 1.3160
Epoch 104/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5952 - loss: 1.2994 - val
_accuracy: 0.5723 - val_loss: 1.3655
Epoch 105/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.5965 - loss: 1.3045 - val
_accuracy: 0.5971 - val_loss: 1.2940
Epoch 106/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5830 - loss: 1.3126 - val
_accuracy: 0.5795 - val_loss: 1.3659
Epoch 107/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5961 - loss: 1.3113 - val
_accuracy: 0.5894 - val_loss: 1.3611
Epoch 108/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.5853 - loss: 1.3377 - val
_accuracy: 0.5888 - val_loss: 1.3384
Epoch 109/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5909 - loss: 1.3150 - val
_accuracy: 0.5904 - val_loss: 1.2922
Epoch 110/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5883 - loss: 1.2977 - val
_accuracy: 0.5883 - val_loss: 1.3373
Epoch 111/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5829 - loss: 1.3291 - val
_accuracy: 0.5888 - val_loss: 1.2926
Epoch 112/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5892 - loss: 1.3245 - val
_accuracy: 0.5837 - val_loss: 1.3450
Epoch 113/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5816 - loss: 1.3314 - val
_accuracy: 0.5816 - val_loss: 1.3199
Epoch 114/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5963 - loss: 1.3409 - val
_accuracy: 0.5739 - val_loss: 1.3434
Epoch 115/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5890 - loss: 1.3046 - val
_accuracy: 0.5888 - val_loss: 1.3112
Epoch 116/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5978 - loss: 1.2959 - val
_accuracy: 0.5847 - val_loss: 1.3416
Epoch 117/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5991 - loss: 1.2944 - val
_accuracy: 0.5687 - val_loss: 1.3624
Epoch 118/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5783 - loss: 1.3459 - val
_accuracy: 0.5790 - val_loss: 1.3345
Epoch 119/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.5867 - loss: 1.3164 - val
_accuracy: 0.6033 - val_loss: 1.2808
Epoch 120/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5918 - loss: 1.3096 - val
_accuracy: 0.5615 - val_loss: 1.3249
```

```
Epoch 121/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5893 - loss: 1.3206 - val
_accuracy: 0.5723 - val_loss: 1.3308
Epoch 122/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5996 - loss: 1.3108 - val
_accuracy: 0.5935 - val_loss: 1.3390
Epoch 123/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5844 - loss: 1.3189 - val
_accuracy: 0.5795 - val_loss: 1.3223
Epoch 124/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6028 - loss: 1.3109 - val
_accuracy: 0.5940 - val_loss: 1.3045
Epoch 125/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5951 - loss: 1.2902 - val
_accuracy: 0.5909 - val_loss: 1.3118
Epoch 126/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5795 - loss: 1.3440 - val
_accuracy: 0.5842 - val_loss: 1.3156
Epoch 127/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5965 - loss: 1.2965 - val
_accuracy: 0.5935 - val_loss: 1.2995
Epoch 128/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5976 - loss: 1.3122 - val
_accuracy: 0.5925 - val_loss: 1.2973
Epoch 129/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5932 - loss: 1.2969 - val
_accuracy: 0.5993 - val_loss: 1.3097
Epoch 130/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5974 - loss: 1.2728 - val
_accuracy: 0.5938 - val_loss: 1.2945
Epoch 131/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5884 - loss: 1.3169 - val
_accuracy: 0.5958 - val_loss: 1.3078
Epoch 132/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.5882 - loss: 1.3160 - val
_accuracy: 0.5739 - val_loss: 1.3332
Epoch 133/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5977 - loss: 1.2889 - val
_accuracy: 0.6012 - val_loss: 1.2851
Epoch 134/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6054 - loss: 1.2619 - val
_accuracy: 0.5940 - val_loss: 1.3197
Epoch 135/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6165 - loss: 1.2698 - val
_accuracy: 0.6007 - val_loss: 1.3560
Epoch 136/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.5958 - loss: 1.3110 - val
_accuracy: 0.5950 - val_loss: 1.2846
Epoch 137/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 64ms/step - accuracy: 0.5968 - loss: 1.2933 - val
_accuracy: 0.5733 - val_loss: 1.3700
Epoch 138/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5961 - loss: 1.2862 - val
_accuracy: 0.5925 - val_loss: 1.2899
Epoch 139/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5821 - loss: 1.3044 - val
_accuracy: 0.5914 - val_loss: 1.2839
Epoch 140/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.5982 - loss: 1.2890 - val
_accuracy: 0.6054 - val_loss: 1.2982
```

```
Epoch 141/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6024 - loss: 1.2700 - val
_accuracy: 0.6054 - val_loss: 1.2818
Epoch 142/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6103 - loss: 1.2804 - val
_accuracy: 0.5728 - val_loss: 1.3556
Epoch 143/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.5968 - loss: 1.3002 - val
_accuracy: 0.5940 - val_loss: 1.2844
Epoch 144/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5955 - loss: 1.2878 - val
_accuracy: 0.5894 - val_loss: 1.3337
Epoch 145/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6053 - loss: 1.2817 - val
_accuracy: 0.5770 - val_loss: 1.3249
Epoch 146/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6151 - loss: 1.2673 - val
_accuracy: 0.6178 - val_loss: 1.2763
Epoch 147/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6119 - loss: 1.2803 - val
_accuracy: 0.6012 - val_loss: 1.3302
Epoch 148/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6026 - loss: 1.2867 - val
_accuracy: 0.6074 - val_loss: 1.2723
Epoch 149/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6038 - loss: 1.2723 - val
_accuracy: 0.6007 - val_loss: 1.2783
Epoch 150/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5978 - loss: 1.2852 - val
_accuracy: 0.5878 - val_loss: 1.3180
Epoch 151/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5976 - loss: 1.2818 - val
_accuracy: 0.5940 - val_loss: 1.2986
Epoch 152/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6037 - loss: 1.2812 - val
_accuracy: 0.6121 - val_loss: 1.2802
Epoch 153/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5913 - loss: 1.3156 - val
_accuracy: 0.5914 - val_loss: 1.3055
Epoch 154/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6047 - loss: 1.2658 - val
_accuracy: 0.6121 - val_loss: 1.2660
Epoch 155/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6121 - loss: 1.2856 - val
_accuracy: 0.6142 - val_loss: 1.2545
Epoch 156/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6102 - loss: 1.2533 - val
_accuracy: 0.5997 - val_loss: 1.3053
Epoch 157/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5998 - loss: 1.2941 - val
_accuracy: 0.6023 - val_loss: 1.3095
Epoch 158/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6019 - loss: 1.2758 - val
_accuracy: 0.5987 - val_loss: 1.2759
Epoch 159/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6014 - loss: 1.2841 - val
_accuracy: 0.5976 - val_loss: 1.3005
Epoch 160/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5905 - loss: 1.2864 - val
_accuracy: 0.6069 - val_loss: 1.2673
```

```
Epoch 161/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6206 - loss: 1.2565 - val
_accuracy: 0.5932 - val_loss: 1.3408
Epoch 162/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.5978 - loss: 1.3130 - val
_accuracy: 0.6043 - val_loss: 1.2782
Epoch 163/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6075 - loss: 1.2647 - val
_accuracy: 0.6084 - val_loss: 1.3071
Epoch 164/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6125 - loss: 1.2686 - val
_accuracy: 0.5806 - val_loss: 1.3218
Epoch 165/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6091 - loss: 1.2630 - val
_accuracy: 0.5909 - val_loss: 1.2872
Epoch 166/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6185 - loss: 1.2584 - val
_accuracy: 0.6033 - val_loss: 1.3117
Epoch 167/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6093 - loss: 1.2618 - val
_accuracy: 0.5873 - val_loss: 1.3170
Epoch 168/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6010 - loss: 1.2770 - val
_accuracy: 0.5868 - val_loss: 1.3239
Epoch 169/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6092 - loss: 1.2686 - val
_accuracy: 0.6018 - val_loss: 1.3052
Epoch 170/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6016 - loss: 1.2866 - val
_accuracy: 0.6002 - val_loss: 1.3193
Epoch 171/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6000 - loss: 1.2936 - val
_accuracy: 0.6136 - val_loss: 1.2826
Epoch 172/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6089 - loss: 1.2659 - val
_accuracy: 0.6209 - val_loss: 1.3111
Epoch 173/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6189 - loss: 1.2689 - val
_accuracy: 0.6069 - val_loss: 1.2937
Epoch 174/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6295 - loss: 1.2190 - val
_accuracy: 0.6240 - val_loss: 1.2325
Epoch 175/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6210 - loss: 1.2552 - val
_accuracy: 0.5981 - val_loss: 1.3117
Epoch 176/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6101 - loss: 1.2521 - val
_accuracy: 0.6126 - val_loss: 1.2935
Epoch 177/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6053 - loss: 1.2849 - val
_accuracy: 0.5795 - val_loss: 1.3518
Epoch 178/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6095 - loss: 1.2747 - val
_accuracy: 0.5981 - val_loss: 1.3184
Epoch 179/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6041 - loss: 1.2578 - val
_accuracy: 0.6002 - val_loss: 1.3164
Epoch 180/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6145 - loss: 1.2402 - val
_accuracy: 0.6049 - val_loss: 1.3255
```

```
Epoch 181/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6256 - loss: 1.2649 - val
_accuracy: 0.5888 - val_loss: 1.3595
Epoch 182/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6045 - loss: 1.2857 - val
_accuracy: 0.6054 - val_loss: 1.2498
Epoch 183/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6152 - loss: 1.2529 - val
_accuracy: 0.6054 - val_loss: 1.2962
Epoch 184/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6141 - loss: 1.2567 - val
_accuracy: 0.5542 - val_loss: 1.4735
Epoch 185/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6155 - loss: 1.2601 - val
_accuracy: 0.6064 - val_loss: 1.3190
Epoch 186/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6164 - loss: 1.2477 - val
_accuracy: 0.6224 - val_loss: 1.2313
Epoch 187/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6047 - loss: 1.2687 - val
_accuracy: 0.6038 - val_loss: 1.3093
Epoch 188/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.5886 - loss: 1.3069 - val
_accuracy: 0.6219 - val_loss: 1.2294
Epoch 189/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6101 - loss: 1.2745 - val
_accuracy: 0.5945 - val_loss: 1.3131
Epoch 190/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6105 - loss: 1.2528 - val
_accuracy: 0.6142 - val_loss: 1.2636
Epoch 191/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6190 - loss: 1.2357 - val
_accuracy: 0.6147 - val_loss: 1.2768
Epoch 192/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6114 - loss: 1.2539 - val
_accuracy: 0.5894 - val_loss: 1.3035
Epoch 193/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6125 - loss: 1.2551 - val
_accuracy: 0.6205 - val_loss: 1.2565
Epoch 194/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6103 - loss: 1.2610 - val
_accuracy: 0.5932 - val_loss: 1.3081
Epoch 195/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6315 - loss: 1.2189 - val
_accuracy: 0.6270 - val_loss: 1.2738
Epoch 196/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6115 - loss: 1.2359 - val
_accuracy: 0.5940 - val_loss: 1.3181
Epoch 197/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6055 - loss: 1.2712 - val
_accuracy: 0.5992 - val_loss: 1.2864
Epoch 198/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6198 - loss: 1.2629 - val
_accuracy: 0.5956 - val_loss: 1.3415
Epoch 199/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6029 - loss: 1.2747 - val
_accuracy: 0.5976 - val_loss: 1.3050
Epoch 200/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6186 - loss: 1.2411 - val
_accuracy: 0.5971 - val_loss: 1.2866
```

```
Epoch 201/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6114 - loss: 1.2621 - val
_accuracy: 0.5925 - val_loss: 1.2994
Epoch 202/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6038 - loss: 1.2495 - val
_accuracy: 0.6095 - val_loss: 1.2953
Epoch 203/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6182 - loss: 1.2567 - val
_accuracy: 0.5940 - val_loss: 1.2724
Epoch 204/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6115 - loss: 1.2591 - val
_accuracy: 0.6105 - val_loss: 1.2571
Epoch 205/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6075 - loss: 1.2577 - val
_accuracy: 0.6028 - val_loss: 1.3002
Epoch 206/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6250 - loss: 1.2396 - val
_accuracy: 0.6157 - val_loss: 1.2401
Epoch 207/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6104 - loss: 1.2829 - val
_accuracy: 0.6080 - val_loss: 1.2658
Epoch 208/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6038 - loss: 1.2770 - val
_accuracy: 0.6157 - val_loss: 1.2601
Epoch 209/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6244 - loss: 1.2381 - val
_accuracy: 0.6116 - val_loss: 1.2502
Epoch 210/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6240 - loss: 1.2575 - val
_accuracy: 0.6064 - val_loss: 1.2663
Epoch 211/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6257 - loss: 1.2203 - val
_accuracy: 0.6188 - val_loss: 1.2445
Epoch 212/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6161 - loss: 1.2624 - val
_accuracy: 0.5873 - val_loss: 1.3676
Epoch 213/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6129 - loss: 1.2490 - val
_accuracy: 0.5997 - val_loss: 1.3117
Epoch 214/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6166 - loss: 1.2479 - val
_accuracy: 0.5966 - val_loss: 1.3448
Epoch 215/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6108 - loss: 1.2561 - val
_accuracy: 0.6209 - val_loss: 1.2555
Epoch 216/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6163 - loss: 1.2498 - val
_accuracy: 0.6007 - val_loss: 1.3320
Epoch 217/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6166 - loss: 1.2430 - val
_accuracy: 0.5976 - val_loss: 1.3250
Epoch 218/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6211 - loss: 1.2316 - val
_accuracy: 0.6100 - val_loss: 1.2670
Epoch 219/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6196 - loss: 1.2427 - val
_accuracy: 0.6069 - val_loss: 1.2979
Epoch 220/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6047 - loss: 1.2605 - val
_accuracy: 0.6023 - val_loss: 1.2863
```

```
Epoch 221/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6243 - loss: 1.2490 - val
_accuracy: 0.6167 - val_loss: 1.2683
Epoch 222/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6158 - loss: 1.2749 - val
_accuracy: 0.6250 - val_loss: 1.2433
Epoch 223/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6160 - loss: 1.2227 - val
_accuracy: 0.5899 - val_loss: 1.3435
Epoch 224/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6248 - loss: 1.2467 - val
_accuracy: 0.6064 - val_loss: 1.2687
Epoch 225/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6282 - loss: 1.2094 - val
_accuracy: 0.6174 - val_loss: 1.2754
Epoch 226/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6223 - loss: 1.2482 - val
_accuracy: 0.6013 - val_loss: 1.2834
Epoch 227/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6179 - loss: 1.2321 - val
_accuracy: 0.5706 - val_loss: 1.4022
Epoch 228/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6365 - loss: 1.2348 - val
_accuracy: 0.6224 - val_loss: 1.2472
Epoch 229/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6161 - loss: 1.2504 - val
_accuracy: 0.6111 - val_loss: 1.2201
Epoch 230/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6224 - loss: 1.2480 - val
_accuracy: 0.6286 - val_loss: 1.3108
Epoch 231/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6254 - loss: 1.2419 - val
_accuracy: 0.6338 - val_loss: 1.2062
Epoch 232/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6091 - loss: 1.2328 - val
_accuracy: 0.6219 - val_loss: 1.2663
Epoch 233/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6215 - loss: 1.2287 - val
_accuracy: 0.6116 - val_loss: 1.2730
Epoch 234/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6080 - loss: 1.2403 - val
_accuracy: 0.6291 - val_loss: 1.2641
Epoch 235/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6283 - loss: 1.2309 - val
_accuracy: 0.6012 - val_loss: 1.3266
Epoch 236/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6232 - loss: 1.2382 - val
_accuracy: 0.6224 - val_loss: 1.2670
Epoch 237/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6132 - loss: 1.2405 - val
_accuracy: 0.6049 - val_loss: 1.3064
Epoch 238/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6108 - loss: 1.2689 - val
_accuracy: 0.6147 - val_loss: 1.2820
Epoch 239/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6221 - loss: 1.2462 - val
_accuracy: 0.6426 - val_loss: 1.2343
Epoch 240/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6263 - loss: 1.2206 - val
_accuracy: 0.5811 - val_loss: 1.3339
```

```
Epoch 241/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6193 - loss: 1.2367 - val
_accuracy: 0.6126 - val_loss: 1.2770
Epoch 242/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6175 - loss: 1.2163 - val
_accuracy: 0.5790 - val_loss: 1.3926
Epoch 243/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6206 - loss: 1.2686 - val
_accuracy: 0.6142 - val_loss: 1.2688
Epoch 244/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6301 - loss: 1.2303 - val
_accuracy: 0.5976 - val_loss: 1.3173
Epoch 245/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6299 - loss: 1.2319 - val
_accuracy: 0.6178 - val_loss: 1.2370
Epoch 246/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6292 - loss: 1.2061 - val
_accuracy: 0.6245 - val_loss: 1.2408
Epoch 247/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6137 - loss: 1.2361 - val
_accuracy: 0.6235 - val_loss: 1.2394
Epoch 248/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 64ms/step - accuracy: 0.6302 - loss: 1.2196 - val
_accuracy: 0.6043 - val_loss: 1.2876
Epoch 249/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6376 - loss: 1.2066 - val
_accuracy: 0.6209 - val_loss: 1.2311
Epoch 250/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6197 - loss: 1.2349 - val
_accuracy: 0.6136 - val_loss: 1.2612
Epoch 251/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6236 - loss: 1.2353 - val
_accuracy: 0.6214 - val_loss: 1.2349
Epoch 252/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6288 - loss: 1.2408 - val
_accuracy: 0.5888 - val_loss: 1.3116
Epoch 253/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6164 - loss: 1.2126 - val
_accuracy: 0.5821 - val_loss: 1.3383
Epoch 254/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6295 - loss: 1.2466 - val
_accuracy: 0.6095 - val_loss: 1.2864
Epoch 255/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6262 - loss: 1.2303 - val
_accuracy: 0.5976 - val_loss: 1.3455
Epoch 256/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6203 - loss: 1.2316 - val
_accuracy: 0.6049 - val_loss: 1.3234
Epoch 257/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6247 - loss: 1.2340 - val
_accuracy: 0.6109 - val_loss: 1.2971
Epoch 258/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 59ms/step - accuracy: 0.6340 - loss: 1.2289 - val
_accuracy: 0.6064 - val_loss: 1.2597
Epoch 259/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6269 - loss: 1.2028 - val
_accuracy: 0.6119 - val_loss: 1.2959
Epoch 260/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6361 - loss: 1.2118 - val
_accuracy: 0.6229 - val_loss: 1.2570
```

```
Epoch 261/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6132 - loss: 1.2498 - val
_accuracy: 0.5708 - val_loss: 1.4374
Epoch 262/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6224 - loss: 1.2394 - val
_accuracy: 0.6198 - val_loss: 1.2520
Epoch 263/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6280 - loss: 1.2100 - val
_accuracy: 0.6028 - val_loss: 1.2946
Epoch 264/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6282 - loss: 1.2239 - val
_accuracy: 0.6219 - val_loss: 1.2664
Epoch 265/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6265 - loss: 1.2181 - val
_accuracy: 0.6136 - val_loss: 1.2647
Epoch 266/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6242 - loss: 1.2432 - val
_accuracy: 0.6131 - val_loss: 1.3090
Epoch 267/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6258 - loss: 1.2333 - val
_accuracy: 0.6121 - val_loss: 1.2669
Epoch 268/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6327 - loss: 1.2117 - val
_accuracy: 0.6023 - val_loss: 1.3231
Epoch 269/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6246 - loss: 1.2352 - val
_accuracy: 0.6235 - val_loss: 1.2315
Epoch 270/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6301 - loss: 1.2307 - val
_accuracy: 0.6214 - val_loss: 1.3042
Epoch 271/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6325 - loss: 1.2172 - val
_accuracy: 0.6183 - val_loss: 1.2582
Epoch 272/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6226 - loss: 1.2482 - val
_accuracy: 0.6157 - val_loss: 1.2950
Epoch 273/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6275 - loss: 1.2228 - val
_accuracy: 0.6085 - val_loss: 1.2911
Epoch 274/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6145 - loss: 1.2390 - val
_accuracy: 0.6023 - val_loss: 1.3071
Epoch 275/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6123 - loss: 1.2368 - val
_accuracy: 0.6312 - val_loss: 1.2252
Epoch 276/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6146 - loss: 1.2451 - val
_accuracy: 0.6214 - val_loss: 1.2578
Epoch 277/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6469 - loss: 1.2012 - val
_accuracy: 0.6033 - val_loss: 1.2960
Epoch 278/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6392 - loss: 1.2011 - val
_accuracy: 0.6173 - val_loss: 1.2460
Epoch 279/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6242 - loss: 1.2291 - val
_accuracy: 0.6147 - val_loss: 1.2779
Epoch 280/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6238 - loss: 1.2324 - val
_accuracy: 0.6136 - val_loss: 1.2621
```

```
Epoch 281/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6288 - loss: 1.2277 - val
_accuracy: 0.6193 - val_loss: 1.2835
Epoch 282/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6319 - loss: 1.2214 - val
_accuracy: 0.6265 - val_loss: 1.2525
Epoch 283/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6361 - loss: 1.1936 - val
_accuracy: 0.6059 - val_loss: 1.2970
Epoch 284/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6329 - loss: 1.2230 - val
_accuracy: 0.6116 - val_loss: 1.2695
Epoch 285/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6305 - loss: 1.2300 - val
_accuracy: 0.6296 - val_loss: 1.2735
Epoch 286/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6315 - loss: 1.2249 - val
_accuracy: 0.6069 - val_loss: 1.2602
Epoch 287/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6223 - loss: 1.2288 - val
_accuracy: 0.6374 - val_loss: 1.2044
Epoch 288/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6375 - loss: 1.2017 - val
_accuracy: 0.6049 - val_loss: 1.2708
Epoch 289/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6405 - loss: 1.2193 - val
_accuracy: 0.6124 - val_loss: 1.2943
Epoch 290/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6321 - loss: 1.2143 - val
_accuracy: 0.5988 - val_loss: 1.2701
Epoch 291/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6178 - loss: 1.2198 - val
_accuracy: 0.6190 - val_loss: 1.2488
Epoch 292/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6258 - loss: 1.2156 - val
_accuracy: 0.6317 - val_loss: 1.2594
Epoch 293/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6333 - loss: 1.2126 - val
_accuracy: 0.6183 - val_loss: 1.2698
Epoch 294/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6338 - loss: 1.2288 - val
_accuracy: 0.5909 - val_loss: 1.3490
Epoch 295/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6331 - loss: 1.2158 - val
_accuracy: 0.5930 - val_loss: 1.3710
Epoch 296/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6401 - loss: 1.2094 - val
_accuracy: 0.6121 - val_loss: 1.2581
Epoch 297/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6375 - loss: 1.1979 - val
_accuracy: 0.6069 - val_loss: 1.2870
Epoch 298/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6370 - loss: 1.2467 - val
_accuracy: 0.6265 - val_loss: 1.2498
Epoch 299/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6313 - loss: 1.2152 - val
_accuracy: 0.6265 - val_loss: 1.2287
Epoch 300/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6356 - loss: 1.2043 - val
_accuracy: 0.6100 - val_loss: 1.3250
```

```
Epoch 301/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 64ms/step - accuracy: 0.6212 - loss: 1.2439 - val_accuracy: 0.6198 - val_loss: 1.2716
Epoch 302/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6338 - loss: 1.2163 - val_accuracy: 0.5945 - val_loss: 1.3091
Epoch 303/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6262 - loss: 1.2384 - val_accuracy: 0.6265 - val_loss: 1.2607
Epoch 304/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6321 - loss: 1.2272 - val_accuracy: 0.6162 - val_loss: 1.2646
Epoch 305/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6369 - loss: 1.1997 - val_accuracy: 0.5971 - val_loss: 1.3013
Epoch 306/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6240 - loss: 1.2325 - val_accuracy: 0.6183 - val_loss: 1.2557
Epoch 307/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6348 - loss: 1.2090 - val_accuracy: 0.6147 - val_loss: 1.2515
Epoch 308/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6307 - loss: 1.2225 - val_accuracy: 0.6100 - val_loss: 1.3019
Epoch 309/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6381 - loss: 1.1828 - val_accuracy: 0.6183 - val_loss: 1.2807
Epoch 310/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6290 - loss: 1.2296 - val_accuracy: 0.6162 - val_loss: 1.2694
Epoch 311/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6438 - loss: 1.1845 - val_accuracy: 0.6167 - val_loss: 1.2891
Epoch 312/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6475 - loss: 1.1837 - val_accuracy: 0.6131 - val_loss: 1.2962
Epoch 313/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6149 - loss: 1.2475 - val_accuracy: 0.6074 - val_loss: 1.3402
Epoch 314/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6251 - loss: 1.2169 - val_accuracy: 0.6136 - val_loss: 1.2518
Epoch 315/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6306 - loss: 1.2113 - val_accuracy: 0.6245 - val_loss: 1.2568
Epoch 316/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6280 - loss: 1.2368 - val_accuracy: 0.6245 - val_loss: 1.2498
Epoch 317/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6368 - loss: 1.1908 - val_accuracy: 0.6255 - val_loss: 1.2695
Epoch 318/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6382 - loss: 1.1824 - val_accuracy: 0.5981 - val_loss: 1.3001
Epoch 319/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6351 - loss: 1.2124 - val_accuracy: 0.6173 - val_loss: 1.2882
Epoch 320/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6311 - loss: 1.2264 - val_accuracy: 0.6105 - val_loss: 1.2694
```

```
Epoch 321/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6351 - loss: 1.2007 - val
_accuracy: 0.6114 - val_loss: 1.2689
Epoch 322/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6331 - loss: 1.2108 - val
_accuracy: 0.6058 - val_loss: 1.2814
Epoch 323/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6271 - loss: 1.2167 - val
_accuracy: 0.6190 - val_loss: 1.2578
Epoch 324/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6423 - loss: 1.1774 - val
_accuracy: 0.6054 - val_loss: 1.3100
Epoch 325/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6366 - loss: 1.1993 - val
_accuracy: 0.6059 - val_loss: 1.3181
Epoch 326/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6375 - loss: 1.1992 - val
_accuracy: 0.6307 - val_loss: 1.2420
Epoch 327/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6294 - loss: 1.2129 - val
_accuracy: 0.5863 - val_loss: 1.3931
Epoch 328/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6230 - loss: 1.2189 - val
_accuracy: 0.6271 - val_loss: 1.2227
Epoch 329/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6312 - loss: 1.2130 - val
_accuracy: 0.6167 - val_loss: 1.2433
Epoch 330/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6408 - loss: 1.2001 - val
_accuracy: 0.6379 - val_loss: 1.2427
Epoch 331/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6348 - loss: 1.2170 - val
_accuracy: 0.6250 - val_loss: 1.2319
Epoch 332/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6237 - loss: 1.2365 - val
_accuracy: 0.6033 - val_loss: 1.3351
Epoch 333/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6304 - loss: 1.2075 - val
_accuracy: 0.6250 - val_loss: 1.2468
Epoch 334/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6274 - loss: 1.2289 - val
_accuracy: 0.6214 - val_loss: 1.2619
Epoch 335/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6471 - loss: 1.1868 - val
_accuracy: 0.6235 - val_loss: 1.2552
Epoch 336/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6425 - loss: 1.1848 - val
_accuracy: 0.6095 - val_loss: 1.2829
Epoch 337/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6380 - loss: 1.1913 - val
_accuracy: 0.5976 - val_loss: 1.2926
Epoch 338/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6396 - loss: 1.1968 - val
_accuracy: 0.6111 - val_loss: 1.3063
Epoch 339/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6225 - loss: 1.2197 - val
_accuracy: 0.6271 - val_loss: 1.2323
Epoch 340/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6457 - loss: 1.1670 - val
_accuracy: 0.6229 - val_loss: 1.2628
```

```
Epoch 341/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6438 - loss: 1.2089 - val
_accuracy: 0.6147 - val_loss: 1.2844
Epoch 342/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6280 - loss: 1.2020 - val
_accuracy: 0.6312 - val_loss: 1.2418
Epoch 343/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - accuracy: 0.6260 - loss: 1.2150 - val
_accuracy: 0.6364 - val_loss: 1.2268
Epoch 344/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6325 - loss: 1.1830 - val
_accuracy: 0.6198 - val_loss: 1.2875
Epoch 345/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6261 - loss: 1.2149 - val
_accuracy: 0.6276 - val_loss: 1.2357
Epoch 346/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6383 - loss: 1.2039 - val
_accuracy: 0.6374 - val_loss: 1.2211
Epoch 347/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6309 - loss: 1.1982 - val
_accuracy: 0.6250 - val_loss: 1.2727
Epoch 348/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6268 - loss: 1.2069 - val
_accuracy: 0.6214 - val_loss: 1.2704
Epoch 349/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6384 - loss: 1.2076 - val
_accuracy: 0.6245 - val_loss: 1.2620
Epoch 350/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6383 - loss: 1.2078 - val
_accuracy: 0.6064 - val_loss: 1.2886
Epoch 351/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6420 - loss: 1.1895 - val
_accuracy: 0.6255 - val_loss: 1.2558
Epoch 352/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6462 - loss: 1.2013 - val
_accuracy: 0.6235 - val_loss: 1.2504
Epoch 353/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6419 - loss: 1.2090 - val
_accuracy: 0.6023 - val_loss: 1.2885
Epoch 354/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6316 - loss: 1.2071 - val
_accuracy: 0.6220 - val_loss: 1.2483
Epoch 355/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6438 - loss: 1.1529 - val
_accuracy: 0.6245 - val_loss: 1.2695
Epoch 356/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6385 - loss: 1.2178 - val
_accuracy: 0.6204 - val_loss: 1.2362
Epoch 357/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6440 - loss: 1.1764 - val
_accuracy: 0.6358 - val_loss: 1.2428
Epoch 358/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6360 - loss: 1.1984 - val
_accuracy: 0.6245 - val_loss: 1.2445
Epoch 359/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6352 - loss: 1.2091 - val
_accuracy: 0.6214 - val_loss: 1.2421
Epoch 360/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6383 - loss: 1.1980 - val
_accuracy: 0.6395 - val_loss: 1.2337
```

```
Epoch 361/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6315 - loss: 1.2137 - val
_accuracy: 0.6338 - val_loss: 1.2358
Epoch 362/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6427 - loss: 1.2107 - val
_accuracy: 0.6173 - val_loss: 1.2619
Epoch 363/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6434 - loss: 1.2141 - val
_accuracy: 0.6240 - val_loss: 1.2495
Epoch 364/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6493 - loss: 1.1830 - val
_accuracy: 0.6007 - val_loss: 1.2883
Epoch 365/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6430 - loss: 1.1918 - val
_accuracy: 0.6147 - val_loss: 1.2965
Epoch 366/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6369 - loss: 1.2162 - val
_accuracy: 0.6209 - val_loss: 1.2559
Epoch 367/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6321 - loss: 1.1919 - val
_accuracy: 0.6420 - val_loss: 1.2259
Epoch 368/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6459 - loss: 1.1734 - val
_accuracy: 0.6173 - val_loss: 1.2693
Epoch 369/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6399 - loss: 1.1729 - val
_accuracy: 0.6286 - val_loss: 1.2506
Epoch 370/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6471 - loss: 1.1816 - val
_accuracy: 0.6095 - val_loss: 1.3708
Epoch 371/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6273 - loss: 1.2335 - val
_accuracy: 0.5945 - val_loss: 1.3691
Epoch 372/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6244 - loss: 1.2089 - val
_accuracy: 0.6012 - val_loss: 1.2859
Epoch 373/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6332 - loss: 1.2038 - val
_accuracy: 0.6281 - val_loss: 1.2208
Epoch 374/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6410 - loss: 1.2066 - val
_accuracy: 0.6173 - val_loss: 1.2805
Epoch 375/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6265 - loss: 1.2162 - val
_accuracy: 0.6162 - val_loss: 1.2636
Epoch 376/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6490 - loss: 1.1822 - val
_accuracy: 0.5945 - val_loss: 1.3469
Epoch 377/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6309 - loss: 1.2069 - val
_accuracy: 0.6167 - val_loss: 1.2774
Epoch 378/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6520 - loss: 1.1577 - val
_accuracy: 0.6018 - val_loss: 1.3144
Epoch 379/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6287 - loss: 1.2117 - val
_accuracy: 0.6317 - val_loss: 1.2785
Epoch 380/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6420 - loss: 1.1918 - val
_accuracy: 0.6002 - val_loss: 1.3091
```

```
Epoch 381/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6394 - loss: 1.1862 - val
_accuracy: 0.6271 - val_loss: 1.2581
Epoch 382/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6428 - loss: 1.1850 - val
_accuracy: 0.6369 - val_loss: 1.2247
Epoch 383/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6303 - loss: 1.2103 - val
_accuracy: 0.6312 - val_loss: 1.2634
Epoch 384/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6493 - loss: 1.1690 - val
_accuracy: 0.6007 - val_loss: 1.3218
Epoch 385/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6396 - loss: 1.1969 - val
_accuracy: 0.6331 - val_loss: 1.2405
Epoch 386/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6380 - loss: 1.2006 - val
_accuracy: 0.6240 - val_loss: 1.2456
Epoch 387/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6463 - loss: 1.1667 - val
_accuracy: 0.6119 - val_loss: 1.2609
Epoch 388/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6292 - loss: 1.1991 - val
_accuracy: 0.6441 - val_loss: 1.2080
Epoch 389/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6495 - loss: 1.1764 - val
_accuracy: 0.6198 - val_loss: 1.2921
Epoch 390/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6493 - loss: 1.1772 - val
_accuracy: 0.6250 - val_loss: 1.2609
Epoch 391/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6428 - loss: 1.1981 - val
_accuracy: 0.6353 - val_loss: 1.2231
Epoch 392/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6394 - loss: 1.1835 - val
_accuracy: 0.5919 - val_loss: 1.3444
Epoch 393/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6422 - loss: 1.1750 - val
_accuracy: 0.6296 - val_loss: 1.2488
Epoch 394/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6418 - loss: 1.1848 - val
_accuracy: 0.6183 - val_loss: 1.2617
Epoch 395/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6496 - loss: 1.2020 - val
_accuracy: 0.6069 - val_loss: 1.2842
Epoch 396/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6481 - loss: 1.1960 - val
_accuracy: 0.6245 - val_loss: 1.2560
Epoch 397/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6418 - loss: 1.1708 - val
_accuracy: 0.6255 - val_loss: 1.2839
Epoch 398/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6411 - loss: 1.1752 - val
_accuracy: 0.6136 - val_loss: 1.2807
Epoch 399/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6435 - loss: 1.1791 - val
_accuracy: 0.6105 - val_loss: 1.2881
Epoch 400/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6435 - loss: 1.1978 - val
_accuracy: 0.6348 - val_loss: 1.2342
```

```
Epoch 401/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6451 - loss: 1.1639 - val
_accuracy: 0.6183 - val_loss: 1.2507
Epoch 402/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6501 - loss: 1.1745 - val
_accuracy: 0.6255 - val_loss: 1.2558
Epoch 403/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6342 - loss: 1.1994 - val
_accuracy: 0.6167 - val_loss: 1.2653
Epoch 404/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6372 - loss: 1.1861 - val
_accuracy: 0.6147 - val_loss: 1.2574
Epoch 405/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6389 - loss: 1.1760 - val
_accuracy: 0.5930 - val_loss: 1.3119
Epoch 406/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6390 - loss: 1.1864 - val
_accuracy: 0.6188 - val_loss: 1.2499
Epoch 407/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6298 - loss: 1.1959 - val
_accuracy: 0.6054 - val_loss: 1.3100
Epoch 408/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6500 - loss: 1.1826 - val
_accuracy: 0.6333 - val_loss: 1.2249
Epoch 409/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6453 - loss: 1.1726 - val
_accuracy: 0.5914 - val_loss: 1.3635
Epoch 410/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6310 - loss: 1.2179 - val
_accuracy: 0.6131 - val_loss: 1.2877
Epoch 411/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6399 - loss: 1.2169 - val
_accuracy: 0.6064 - val_loss: 1.2957
Epoch 412/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6410 - loss: 1.1965 - val
_accuracy: 0.6219 - val_loss: 1.2573
Epoch 413/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6448 - loss: 1.1991 - val
_accuracy: 0.6322 - val_loss: 1.2150
Epoch 414/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6376 - loss: 1.1854 - val
_accuracy: 0.6204 - val_loss: 1.2903
Epoch 415/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6413 - loss: 1.2081 - val
_accuracy: 0.6152 - val_loss: 1.2662
Epoch 416/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6362 - loss: 1.1903 - val
_accuracy: 0.6250 - val_loss: 1.2655
Epoch 417/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6493 - loss: 1.1840 - val
_accuracy: 0.5862 - val_loss: 1.3619
Epoch 418/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 66ms/step - accuracy: 0.6439 - loss: 1.2092 - val
_accuracy: 0.6149 - val_loss: 1.2937
Epoch 419/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 66ms/step - accuracy: 0.6311 - loss: 1.1991 - val
_accuracy: 0.6421 - val_loss: 1.2520
Epoch 420/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6345 - loss: 1.2264 - val
_accuracy: 0.6276 - val_loss: 1.2739
```

```
Epoch 421/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6358 - loss: 1.1904 - val
_accuracy: 0.6260 - val_loss: 1.2607
Epoch 422/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6494 - loss: 1.1816 - val
_accuracy: 0.6327 - val_loss: 1.2327
Epoch 423/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 65ms/step - accuracy: 0.6303 - loss: 1.2155 - val
_accuracy: 0.6142 - val_loss: 1.2411
Epoch 424/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6310 - loss: 1.1943 - val
_accuracy: 0.6214 - val_loss: 1.2555
Epoch 425/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6503 - loss: 1.1836 - val
_accuracy: 0.6167 - val_loss: 1.2552
Epoch 426/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6371 - loss: 1.2022 - val
_accuracy: 0.6271 - val_loss: 1.2839
Epoch 427/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6400 - loss: 1.1838 - val
_accuracy: 0.6276 - val_loss: 1.2555
Epoch 428/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6389 - loss: 1.2058 - val
_accuracy: 0.6276 - val_loss: 1.2748
Epoch 429/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6529 - loss: 1.1868 - val
_accuracy: 0.6441 - val_loss: 1.2314
Epoch 430/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6386 - loss: 1.1789 - val
_accuracy: 0.6302 - val_loss: 1.2582
Epoch 431/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6447 - loss: 1.2071 - val
_accuracy: 0.6018 - val_loss: 1.3634
Epoch 432/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6311 - loss: 1.2255 - val
_accuracy: 0.6224 - val_loss: 1.2580
Epoch 433/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6349 - loss: 1.1873 - val
_accuracy: 0.6126 - val_loss: 1.3106
Epoch 434/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6451 - loss: 1.1920 - val
_accuracy: 0.6157 - val_loss: 1.2495
Epoch 435/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6397 - loss: 1.2156 - val
_accuracy: 0.6224 - val_loss: 1.2876
Epoch 436/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6594 - loss: 1.1752 - val
_accuracy: 0.6059 - val_loss: 1.3113
Epoch 437/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6386 - loss: 1.1624 - val
_accuracy: 0.6322 - val_loss: 1.2233
Epoch 438/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6426 - loss: 1.1835 - val
_accuracy: 0.6080 - val_loss: 1.3285
Epoch 439/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6491 - loss: 1.1735 - val
_accuracy: 0.6235 - val_loss: 1.2467
Epoch 440/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6515 - loss: 1.1853 - val
_accuracy: 0.6271 - val_loss: 1.2621
```

```
Epoch 441/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6490 - loss: 1.1803 - val
_accuracy: 0.6209 - val_loss: 1.2948
Epoch 442/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6395 - loss: 1.2081 - val
_accuracy: 0.6183 - val_loss: 1.2763
Epoch 443/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6356 - loss: 1.2152 - val
_accuracy: 0.6167 - val_loss: 1.2701
Epoch 444/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6486 - loss: 1.1870 - val
_accuracy: 0.5764 - val_loss: 1.4278
Epoch 445/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6384 - loss: 1.1911 - val
_accuracy: 0.6069 - val_loss: 1.3011
Epoch 446/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6168 - loss: 1.2029 - val
_accuracy: 0.6152 - val_loss: 1.2811
Epoch 447/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6509 - loss: 1.1707 - val
_accuracy: 0.6255 - val_loss: 1.2440
Epoch 448/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6454 - loss: 1.1890 - val
_accuracy: 0.5857 - val_loss: 1.3583
Epoch 449/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6410 - loss: 1.2111 - val
_accuracy: 0.6089 - val_loss: 1.2798
Epoch 450/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6445 - loss: 1.1712 - val
_accuracy: 0.6210 - val_loss: 1.2591
Epoch 451/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6333 - loss: 1.2115 - val
_accuracy: 0.6295 - val_loss: 1.2636
Epoch 452/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6476 - loss: 1.1835 - val
_accuracy: 0.6317 - val_loss: 1.2699
Epoch 453/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6429 - loss: 1.1995 - val
_accuracy: 0.6250 - val_loss: 1.2913
Epoch 454/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6386 - loss: 1.2096 - val
_accuracy: 0.6245 - val_loss: 1.2466
Epoch 455/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6444 - loss: 1.1910 - val
_accuracy: 0.6312 - val_loss: 1.2532
Epoch 456/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6492 - loss: 1.1604 - val
_accuracy: 0.6224 - val_loss: 1.2746
Epoch 457/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6527 - loss: 1.1688 - val
_accuracy: 0.6240 - val_loss: 1.2272
Epoch 458/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6516 - loss: 1.1688 - val
_accuracy: 0.6369 - val_loss: 1.2375
Epoch 459/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6480 - loss: 1.1704 - val
_accuracy: 0.6157 - val_loss: 1.2880
Epoch 460/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6499 - loss: 1.1954 - val
_accuracy: 0.6322 - val_loss: 1.2402
```

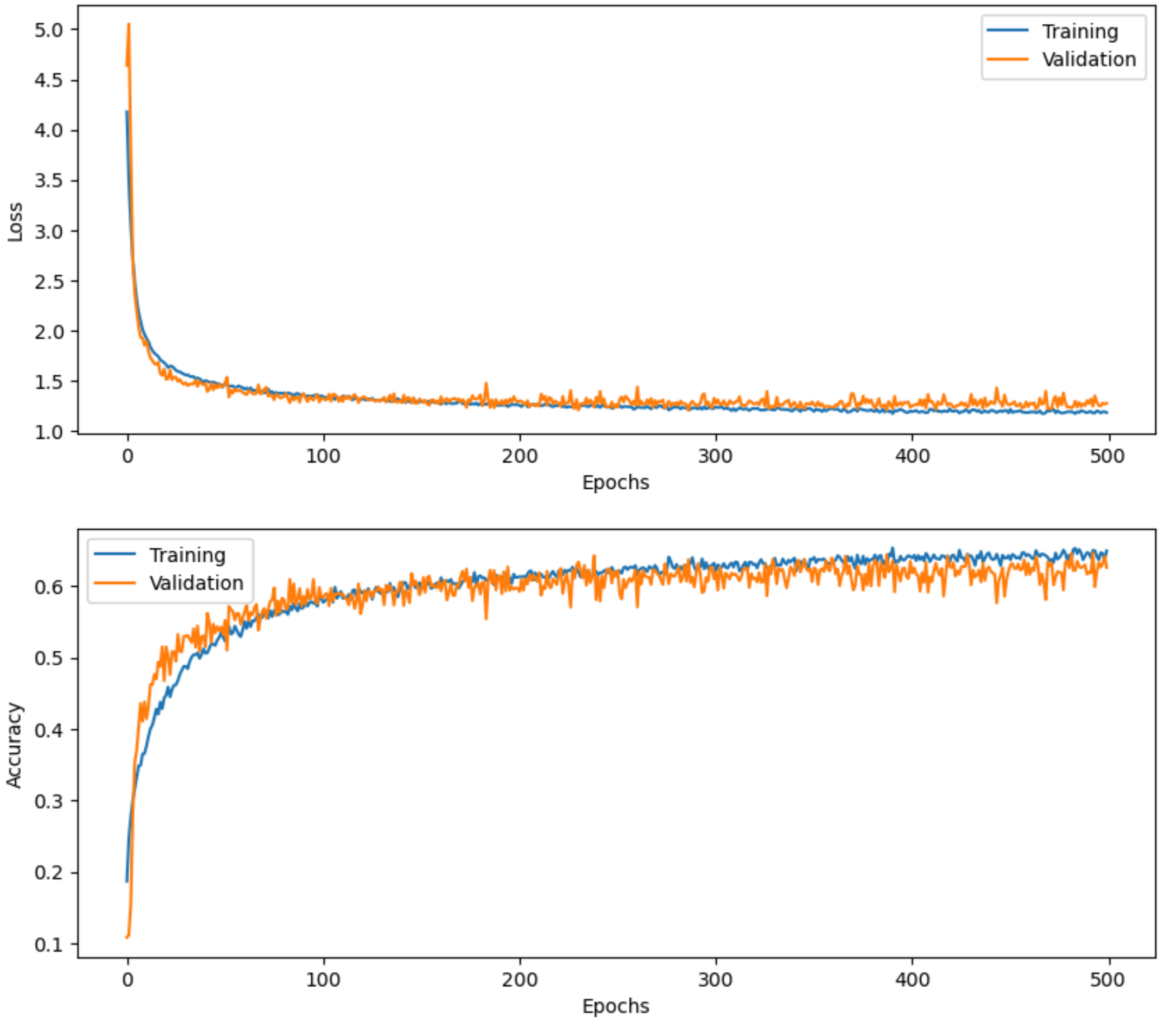
```
Epoch 461/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6507 - loss: 1.1683 - val
_accuracy: 0.6167 - val_loss: 1.2748
Epoch 462/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6394 - loss: 1.1922 - val
_accuracy: 0.6255 - val_loss: 1.2672
Epoch 463/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6362 - loss: 1.2005 - val
_accuracy: 0.6219 - val_loss: 1.2804
Epoch 464/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6456 - loss: 1.1825 - val
_accuracy: 0.6415 - val_loss: 1.2191
Epoch 465/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6412 - loss: 1.1805 - val
_accuracy: 0.6286 - val_loss: 1.2390
Epoch 466/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6515 - loss: 1.1748 - val
_accuracy: 0.6023 - val_loss: 1.3250
Epoch 467/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6409 - loss: 1.1815 - val
_accuracy: 0.6173 - val_loss: 1.2632
Epoch 468/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6598 - loss: 1.1358 - val
_accuracy: 0.6049 - val_loss: 1.2758
Epoch 469/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6425 - loss: 1.1831 - val
_accuracy: 0.5811 - val_loss: 1.3934
Epoch 470/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6428 - loss: 1.2043 - val
_accuracy: 0.6338 - val_loss: 1.2226
Epoch 471/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6380 - loss: 1.1767 - val
_accuracy: 0.6286 - val_loss: 1.2235
Epoch 472/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6482 - loss: 1.1868 - val
_accuracy: 0.6322 - val_loss: 1.2179
Epoch 473/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 60ms/step - accuracy: 0.6407 - loss: 1.1926 - val
_accuracy: 0.6018 - val_loss: 1.3116
Epoch 474/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 64ms/step - accuracy: 0.6462 - loss: 1.1754 - val
_accuracy: 0.6126 - val_loss: 1.3427
Epoch 475/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6352 - loss: 1.2040 - val
_accuracy: 0.6204 - val_loss: 1.2377
Epoch 476/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6406 - loss: 1.1764 - val
_accuracy: 0.6193 - val_loss: 1.3292
Epoch 477/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6464 - loss: 1.1673 - val
_accuracy: 0.6188 - val_loss: 1.2571
Epoch 478/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6525 - loss: 1.1706 - val
_accuracy: 0.5945 - val_loss: 1.3763
Epoch 479/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6467 - loss: 1.1706 - val
_accuracy: 0.6338 - val_loss: 1.2755
Epoch 480/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6445 - loss: 1.1705 - val
_accuracy: 0.6353 - val_loss: 1.2357
```

```
Epoch 481/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6430 - loss: 1.1907 - val
_accuracy: 0.6255 - val_loss: 1.2384
Epoch 482/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6291 - loss: 1.1976 - val
_accuracy: 0.6457 - val_loss: 1.2271
Epoch 483/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6553 - loss: 1.1603 - val
_accuracy: 0.6220 - val_loss: 1.2431
Epoch 484/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6581 - loss: 1.1518 - val
_accuracy: 0.6219 - val_loss: 1.2952
Epoch 485/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6589 - loss: 1.1678 - val
_accuracy: 0.6348 - val_loss: 1.2370
Epoch 486/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 63ms/step - accuracy: 0.6456 - loss: 1.1873 - val
_accuracy: 0.6281 - val_loss: 1.2839
Epoch 487/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6558 - loss: 1.1729 - val
_accuracy: 0.6255 - val_loss: 1.2418
Epoch 488/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6310 - loss: 1.2018 - val
_accuracy: 0.6276 - val_loss: 1.2705
Epoch 489/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6479 - loss: 1.1571 - val
_accuracy: 0.6296 - val_loss: 1.2515
Epoch 490/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6426 - loss: 1.1899 - val
_accuracy: 0.6167 - val_loss: 1.3016
Epoch 491/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6530 - loss: 1.1889 - val
_accuracy: 0.6198 - val_loss: 1.2725
Epoch 492/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6461 - loss: 1.1872 - val
_accuracy: 0.6183 - val_loss: 1.3175
Epoch 493/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6463 - loss: 1.1780 - val
_accuracy: 0.6467 - val_loss: 1.2455
Epoch 494/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6552 - loss: 1.1611 - val
_accuracy: 0.5992 - val_loss: 1.3490
Epoch 495/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6436 - loss: 1.1934 - val
_accuracy: 0.6209 - val_loss: 1.2874
Epoch 496/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6343 - loss: 1.1859 - val
_accuracy: 0.6333 - val_loss: 1.2509
Epoch 497/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6426 - loss: 1.1799 - val
_accuracy: 0.6265 - val_loss: 1.2480
Epoch 498/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 61ms/step - accuracy: 0.6532 - loss: 1.1831 - val
_accuracy: 0.6317 - val_loss: 1.2775
Epoch 499/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 64ms/step - accuracy: 0.6405 - loss: 1.1847 - val
_accuracy: 0.6420 - val_loss: 1.2665
Epoch 500/500
125/125 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.6549 - loss: 1.1771 - val
_accuracy: 0.6260 - val_loss: 1.2740
```

In [31]:
```
# Check if there is still a big difference in accuracy for original and rotated
Xtest_rotated = myrotate(Xtest)
# Evaluate the trained model on original test set
score = model6.evaluate(Xtest, Ytest, batch_size = batch_size, verbose=0)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

# Evaluate the trained model on rotated test set
score = model6.evaluate(Xtest_rotated, Ytest, batch_size = batch_size, verbose=0
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
Test loss: 1.3711
Test accuracy: 0.6175
Test loss: 2.6332
Test accuracy: 0.3230
```

In [32]:
```
# Plot the history from the training run
from utilities import plot_results
plot_results(history6)
```



Plot misclassified images

Lets plot some images where the CNN performed badly.

In [33]:
```
# Find misclassified images
y_pred=model6.predict(Xtest, verbose=0)
```

```
y_pred=np.argmax(y_pred,axis=1)

y_correct = np.argmax(Ytest,axis=-1)

miss = np.flatnonzero(y_correct != y_pred)
```

In [34]:
```
# Plot a few of them
plt.figure(figsize=(15,4))
perm = np.random.permutation(miss)
for i in range(18):
    im = (Xtest[perm[i]] + 1) * 127.5
    im = im.astype('int')
    label_correct = y_correct[perm[i]]
    label_pred = y_pred[perm[i]]

    plt.subplot(3,6,i+1)
    plt.tight_layout()
    plt.imshow(im)
    plt.axis('off')
    plt.title("{}, classified as {}".format(classes[label_correct], classes[labe
plt.show()
```

5.3 Testing on another size

Questions

28. This CNN has been trained on 32 x 32 images, can it be applied to images of another size? If not, why is this the case?

29. Is it possible to design a CNN that can be trained on images of one size, and then applied to an image of any size? How?

Answers

28. Not really - if one tries to feed it an image of a different size, the dimensions of the feature maps will change, causing a mismatch in the input size to the dense layers and resulting in an error.

29. Yes, for instance, using GlobalAveragePooling2D or GlobalMaxPooling2D as the pooling layer.

Part 6: Carbon footprint

In this next section we will evaluate the carbon footprint of training our CNN model. In particular we will look at the effect of training hyper parameters of carbon footprint. You can read more about this topic here or here.

In this lab we will use the `carbontracker` library that easily integrates with any model training routine. See the example in the documentation on how to use the carbon tracker.

Questions

28. Keeping the model architecture fixed, which training parameter impacts the carbon footprint?

29. The choice of batch size can dramatically impact carbon foot print: why is this the case?

30. Assume that you have a model with 100 million parameters running in the backend of a service with 5 million users. How can the carbon footprint of using this model be reduced?

Answers

28, 29. Batch size. A smaller batch size requires more iterations per epoch to process the same number of samples, increasing energy consumption. Although on this system and with this model the footprint is minimal (near zero) even if batch size is 1 (see below).

30. Probably using a simpler model that approximates the original model performance but has fewer parameters. Using transfer learning. Choosing locations with greener data centers, like Iceland or Paraguay. Using energy-efficient hardware.

In [36]:
```
from carbontracker.tracker import CarbonTracker

# --------------------------------------------
# === Your code here ========================
# --------------------------------------------
# Setup training parameters
batch_size = 1
epochs = 10
input_shape = input_shape

# Build model (your best config)
model7 = build_CNN(input_shape=input_shape,
                   loss=CategoricalCrossentropy(),
                   learning_rate=0.0005,
                   n_dense_layers=1,
                   n_conv_layers=3,
                   use_dropout=True,
                   l2_regularization=True,
                   optimizer="adam"
)

# Create a CarbonTracker object
```

```
tracker = CarbonTracker(epochs=epochs)

# start carbon tracking
tracker.epoch_start()

# fit model
model.fit(Xtrain, Ytrain, validation_data=(Xval, Yval), epochs=epochs, batch_siz

tracker.epoch_end()

# ============================================
```

```
CarbonTracker: The following components were found: GPU with device(s) NVIDIA GeF
orce RTX 2060.
Epoch 1/10
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to average
carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to avera
ge carbon intensity 40.694878 gCO2/kWh.
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 26s 4ms/step - accuracy: 0.3143 - loss: 2.6238 - v
al_accuracy: 0.3860 - val_loss: 1.7085
Epoch 2/10
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.4805 - loss: 1.4788 - v
al_accuracy: 0.4207 - val_loss: 1.6811
Epoch 3/10
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.5529 - loss: 1.2910 - v
al_accuracy: 0.4373 - val_loss: 1.6969
Epoch 4/10
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.6108 - loss: 1.1395 - v
al_accuracy: 0.4260 - val_loss: 1.6933
Epoch 5/10
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.6438 - loss: 1.0358 - v
al_accuracy: 0.4433 - val_loss: 1.6502
Epoch 6/10
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.6824 - loss: 0.9328 - v
al_accuracy: 0.4427 - val_loss: 1.7612
Epoch 7/10
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.7105 - loss: 0.8370 - v
al_accuracy: 0.4393 - val_loss: 1.9084
Epoch 8/10
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.7430 - loss: 0.7572 - v
al_accuracy: 0.4700 - val_loss: 2.0591
Epoch 9/10
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.7707 - loss: 0.6814 - v
al_accuracy: 0.4533 - val_loss: 2.1347
Epoch 10/10
7000/7000 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.7701 - loss: 0.6603 - v
al_accuracy: 0.4193 - val_loss: 2.3089
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to average
carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to avera
ge carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: Live carbon intensity could not be fetched at detected location: L
inköping, Östergötland, SE. Defaulted to average carbon intensity for SE in 2023
of 40.69 gCO2/kWh. at detected location: Linköping, Östergötland, SE.
CarbonTracker:
Predicted consumption for 10 epoch(s):
        Time:   0:42:11
        Energy: 0.004779051710 kWh
        CO2eq:  0.194482926300 g
        This is equivalent to:
        0.001809143500 km travelled by car
```

Part 7: Pre-trained 2D CNNs

There are many deep 2D CNNs that have been pre-trained using the large ImageNet database (several million images, 1000 classes). Import a pre-trained ResNet50 network from Keras applications. Show the network using `model.summary()`

Questions

31. How many convolutional layers does ResNet50 have?

32. How many trainable parameters does the ResNet50 network have?

33. What is the size of the images that ResNet50 expects as input?

34. Using the answer to question 30, explain why the second derivative is seldom used when training deep networks.

35. What do you expect the carbon footprint of using pre-trained networks to be compared to training a model from scratch?

Answers

31. 49 convolutional layers
32. 25583592 parameters
33. 224x224 pixels with 3 color channels (RGB)
34. The second derivative is rarely used in deep network training because of huge computational cost and memory cost, higher carbon footprint and efficiency of optimization methods that use only first derivatives (SGD, Adam etc)
35. There will be much lower carbon footprint because networks are pretrained and one needs only fine-tuning.

After loading the pre-trained CNN, apply it to 5 random color images that you download and copy to the cloud machine or your own computer. Are the predictions correct? How certain is the network of each image class?

These pre-trained networks can be fine tuned to your specific data, and normally only the last layers need to be re-trained, but it will still be too time consuming to do in this elaboration.

Some useful functions:

- `load_img` and `img_to_array` in tf_keras.utils.
- `ResNet50` in tf_keras.applications.ResNet50.
- `preprocess_input` in tf_keras.applications.resnet.
- `decode_predictions` in tf_keras.applications.resnet.
- `expand_dims` in numpy.

See keras applications and the keras resnet50-function for more details.

In [40]:
```
# --------------------------------------------
# === Your code here ========================
# --------------------------------------------
# import the necessary libraries and functions
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.utils import load_img, img_to_array
from tensorflow.keras.applications.resnet import preprocess_input, decode_predic

# load the pre-trained ResNet50 model
```

```python
resnet50 = ResNet50(weights="imagenet")

# print the model summary
resnet50.summary()

# Load the image and preprocess it
image = load_img("cat1.jpg", target_size=(224, 224))
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
image = preprocess_input(image)

# predict the image
preds = resnet50.predict(image)
label = decode_predictions(preds, top=1)[0]

# print the predicted label
print(label)

# Load the image and preprocess it
image = load_img("cat2.jpg", target_size=(224, 224))
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
image = preprocess_input(image)

# predict the image
preds = resnet50.predict(image)
label = decode_predictions(preds, top=1)[0]

# print the predicted label
print(label)

# Load the image and preprocess it
image = load_img("cat3.jpg", target_size=(224, 224))
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
image = preprocess_input(image)

# predict the image
preds = resnet50.predict(image)
label = decode_predictions(preds, top=1)[0]

# print the predicted label
print(label)

# Load the image and preprocess it
image = load_img("cat4.jpg", target_size=(224, 224))
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
image = preprocess_input(image)

# predict the image
preds = resnet50.predict(image)
label = decode_predictions(preds, top=1)[0]

# print the predicted label
print(label)

# Load the image and preprocess it
image = load_img("cat5.jpg", target_size=(224, 224))
image = img_to_array(image)
```

```
image = np.expand_dims(image, axis=0)
image = preprocess_input(image)

# predict the image
preds = resnet50.predict(image)
label = decode_predictions(preds, top=1)[0]

# print the predicted label
print(label)

# ============================================
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-application
s/resnet/resnet50_weights_tf_dim_ordering_tf_kernels.h5
102967424/102967424 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step
```

Model: "resnet50"

Layer (type)	Output Shape	Para
input_layer_8 (InputLayer)	(None, 224, 224, 3)	
conv1_pad (ZeroPadding2D)	(None, 230, 230, 3)	
conv1_conv (Conv2D)	(None, 112, 112, 64)	9,
conv1_bn (BatchNormalization)	(None, 112, 112, 64)	
conv1_relu (Activation)	(None, 112, 112, 64)	
pool1_pad (ZeroPadding2D)	(None, 114, 114, 64)	
pool1_pool (MaxPooling2D)	(None, 56, 56, 64)	
conv2_block1_1_conv (Conv2D)	(None, 56, 56, 64)	4,
conv2_block1_1_bn (BatchNormalization)	(None, 56, 56, 64)	
conv2_block1_1_relu (Activation)	(None, 56, 56, 64)	
conv2_block1_2_conv (Conv2D)	(None, 56, 56, 64)	36,
conv2_block1_2_bn (BatchNormalization)	(None, 56, 56, 64)	
conv2_block1_2_relu (Activation)	(None, 56, 56, 64)	
conv2_block1_0_conv (Conv2D)	(None, 56, 56, 256)	16,
conv2_block1_3_conv (Conv2D)	(None, 56, 56, 256)	16,
conv2_block1_0_bn (BatchNormalization)	(None, 56, 56, 256)	1,
conv2_block1_3_bn (BatchNormalization)	(None, 56, 56, 256)	1,
conv2_block1_add (Add)	(None, 56, 56, 256)	
conv2_block1_out (Activation)	(None, 56, 56, 256)	
conv2_block2_1_conv (Conv2D)	(None, 56, 56, 64)	16,
conv2_block2_1_bn (BatchNormalization)	(None, 56, 56, 64)	
conv2_block2_1_relu (Activation)	(None, 56, 56, 64)	
conv2_block2_2_conv (Conv2D)	(None, 56, 56, 64)	36,
conv2_block2_2_bn (BatchNormalization)	(None, 56, 56, 64)	

conv2_block2_2_relu (Activation)	(None, 56, 56, 64)	
conv2_block2_3_conv (Conv2D)	(None, 56, 56, 256)	16,
conv2_block2_3_bn (BatchNormalization)	(None, 56, 56, 256)	1,
conv2_block2_add (Add)	(None, 56, 56, 256)	
conv2_block2_out (Activation)	(None, 56, 56, 256)	
conv2_block3_1_conv (Conv2D)	(None, 56, 56, 64)	16,
conv2_block3_1_bn (BatchNormalization)	(None, 56, 56, 64)	
conv2_block3_1_relu (Activation)	(None, 56, 56, 64)	
conv2_block3_2_conv (Conv2D)	(None, 56, 56, 64)	36,
conv2_block3_2_bn (BatchNormalization)	(None, 56, 56, 64)	
conv2_block3_2_relu (Activation)	(None, 56, 56, 64)	
conv2_block3_3_conv (Conv2D)	(None, 56, 56, 256)	16,
conv2_block3_3_bn (BatchNormalization)	(None, 56, 56, 256)	1,
conv2_block3_add (Add)	(None, 56, 56, 256)	
conv2_block3_out (Activation)	(None, 56, 56, 256)	
conv3_block1_1_conv (Conv2D)	(None, 28, 28, 128)	32,
conv3_block1_1_bn (BatchNormalization)	(None, 28, 28, 128)	
conv3_block1_1_relu (Activation)	(None, 28, 28, 128)	
conv3_block1_2_conv (Conv2D)	(None, 28, 28, 128)	147,
conv3_block1_2_bn (BatchNormalization)	(None, 28, 28, 128)	
conv3_block1_2_relu (Activation)	(None, 28, 28, 128)	
conv3_block1_0_conv (Conv2D)	(None, 28, 28, 512)	131,
conv3_block1_3_conv (Conv2D)	(None, 28, 28, 512)	66,

conv3_block1_0_bn (BatchNormalization)	(None, 28, 28, 512)	2,
conv3_block1_3_bn (BatchNormalization)	(None, 28, 28, 512)	2,
conv3_block1_add (Add)	(None, 28, 28, 512)	
conv3_block1_out (Activation)	(None, 28, 28, 512)	
conv3_block2_1_conv (Conv2D)	(None, 28, 28, 128)	65,
conv3_block2_1_bn (BatchNormalization)	(None, 28, 28, 128)	
conv3_block2_1_relu (Activation)	(None, 28, 28, 128)	
conv3_block2_2_conv (Conv2D)	(None, 28, 28, 128)	147,
conv3_block2_2_bn (BatchNormalization)	(None, 28, 28, 128)	
conv3_block2_2_relu (Activation)	(None, 28, 28, 128)	
conv3_block2_3_conv (Conv2D)	(None, 28, 28, 512)	66,
conv3_block2_3_bn (BatchNormalization)	(None, 28, 28, 512)	2,
conv3_block2_add (Add)	(None, 28, 28, 512)	
conv3_block2_out (Activation)	(None, 28, 28, 512)	
conv3_block3_1_conv (Conv2D)	(None, 28, 28, 128)	65,
conv3_block3_1_bn (BatchNormalization)	(None, 28, 28, 128)	
conv3_block3_1_relu (Activation)	(None, 28, 28, 128)	
conv3_block3_2_conv (Conv2D)	(None, 28, 28, 128)	147,
conv3_block3_2_bn (BatchNormalization)	(None, 28, 28, 128)	
conv3_block3_2_relu (Activation)	(None, 28, 28, 128)	
conv3_block3_3_conv (Conv2D)	(None, 28, 28, 512)	66,
conv3_block3_3_bn (BatchNormalization)	(None, 28, 28, 512)	2,
conv3_block3_add (Add)	(None, 28, 28, 512)	

conv3_block3_out (Activation)	(None, 28, 28, 512)	
conv3_block4_1_conv (Conv2D)	(None, 28, 28, 128)	65,
conv3_block4_1_bn (BatchNormalization)	(None, 28, 28, 128)	
conv3_block4_1_relu (Activation)	(None, 28, 28, 128)	
conv3_block4_2_conv (Conv2D)	(None, 28, 28, 128)	147,
conv3_block4_2_bn (BatchNormalization)	(None, 28, 28, 128)	
conv3_block4_2_relu (Activation)	(None, 28, 28, 128)	
conv3_block4_3_conv (Conv2D)	(None, 28, 28, 512)	66,
conv3_block4_3_bn (BatchNormalization)	(None, 28, 28, 512)	2,
conv3_block4_add (Add)	(None, 28, 28, 512)	
conv3_block4_out (Activation)	(None, 28, 28, 512)	
conv4_block1_1_conv (Conv2D)	(None, 14, 14, 256)	131,
conv4_block1_1_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block1_1_relu (Activation)	(None, 14, 14, 256)	
conv4_block1_2_conv (Conv2D)	(None, 14, 14, 256)	590,
conv4_block1_2_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block1_2_relu (Activation)	(None, 14, 14, 256)	
conv4_block1_0_conv (Conv2D)	(None, 14, 14, 1024)	525,
conv4_block1_3_conv (Conv2D)	(None, 14, 14, 1024)	263,
conv4_block1_0_bn (BatchNormalization)	(None, 14, 14, 1024)	4,
conv4_block1_3_bn (BatchNormalization)	(None, 14, 14, 1024)	4,
conv4_block1_add (Add)	(None, 14, 14, 1024)	
conv4_block1_out (Activation)	(None, 14, 14, 1024)	

conv4_block2_1_conv (Conv2D)	(None, 14, 14, 256)	262,
conv4_block2_1_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block2_1_relu (Activation)	(None, 14, 14, 256)	
conv4_block2_2_conv (Conv2D)	(None, 14, 14, 256)	590,
conv4_block2_2_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block2_2_relu (Activation)	(None, 14, 14, 256)	
conv4_block2_3_conv (Conv2D)	(None, 14, 14, 1024)	263,
conv4_block2_3_bn (BatchNormalization)	(None, 14, 14, 1024)	4,
conv4_block2_add (Add)	(None, 14, 14, 1024)	
conv4_block2_out (Activation)	(None, 14, 14, 1024)	
conv4_block3_1_conv (Conv2D)	(None, 14, 14, 256)	262,
conv4_block3_1_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block3_1_relu (Activation)	(None, 14, 14, 256)	
conv4_block3_2_conv (Conv2D)	(None, 14, 14, 256)	590,
conv4_block3_2_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block3_2_relu (Activation)	(None, 14, 14, 256)	
conv4_block3_3_conv (Conv2D)	(None, 14, 14, 1024)	263,
conv4_block3_3_bn (BatchNormalization)	(None, 14, 14, 1024)	4,
conv4_block3_add (Add)	(None, 14, 14, 1024)	
conv4_block3_out (Activation)	(None, 14, 14, 1024)	
conv4_block4_1_conv (Conv2D)	(None, 14, 14, 256)	262,
conv4_block4_1_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block4_1_relu (Activation)	(None, 14, 14, 256)	

conv4_block4_2_conv (Conv2D)	(None, 14, 14, 256)	590,
conv4_block4_2_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block4_2_relu (Activation)	(None, 14, 14, 256)	
conv4_block4_3_conv (Conv2D)	(None, 14, 14, 1024)	263,
conv4_block4_3_bn (BatchNormalization)	(None, 14, 14, 1024)	4,
conv4_block4_add (Add)	(None, 14, 14, 1024)	
conv4_block4_out (Activation)	(None, 14, 14, 1024)	
conv4_block5_1_conv (Conv2D)	(None, 14, 14, 256)	262,
conv4_block5_1_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block5_1_relu (Activation)	(None, 14, 14, 256)	
conv4_block5_2_conv (Conv2D)	(None, 14, 14, 256)	590,
conv4_block5_2_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block5_2_relu (Activation)	(None, 14, 14, 256)	
conv4_block5_3_conv (Conv2D)	(None, 14, 14, 1024)	263,
conv4_block5_3_bn (BatchNormalization)	(None, 14, 14, 1024)	4,
conv4_block5_add (Add)	(None, 14, 14, 1024)	
conv4_block5_out (Activation)	(None, 14, 14, 1024)	
conv4_block6_1_conv (Conv2D)	(None, 14, 14, 256)	262,
conv4_block6_1_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block6_1_relu (Activation)	(None, 14, 14, 256)	
conv4_block6_2_conv (Conv2D)	(None, 14, 14, 256)	590,
conv4_block6_2_bn (BatchNormalization)	(None, 14, 14, 256)	1,
conv4_block6_2_relu (Activation)	(None, 14, 14, 256)	

conv4_block6_3_conv (Conv2D)	(None, 14, 14, 1024)	263,
conv4_block6_3_bn (BatchNormalization)	(None, 14, 14, 1024)	4,
conv4_block6_add (Add)	(None, 14, 14, 1024)	
conv4_block6_out (Activation)	(None, 14, 14, 1024)	
conv5_block1_1_conv (Conv2D)	(None, 7, 7, 512)	524,
conv5_block1_1_bn (BatchNormalization)	(None, 7, 7, 512)	2,
conv5_block1_1_relu (Activation)	(None, 7, 7, 512)	
conv5_block1_2_conv (Conv2D)	(None, 7, 7, 512)	2,359,
conv5_block1_2_bn (BatchNormalization)	(None, 7, 7, 512)	2,
conv5_block1_2_relu (Activation)	(None, 7, 7, 512)	
conv5_block1_0_conv (Conv2D)	(None, 7, 7, 2048)	2,099,
conv5_block1_3_conv (Conv2D)	(None, 7, 7, 2048)	1,050,
conv5_block1_0_bn (BatchNormalization)	(None, 7, 7, 2048)	8,
conv5_block1_3_bn (BatchNormalization)	(None, 7, 7, 2048)	8,
conv5_block1_add (Add)	(None, 7, 7, 2048)	
conv5_block1_out (Activation)	(None, 7, 7, 2048)	
conv5_block2_1_conv (Conv2D)	(None, 7, 7, 512)	1,049,
conv5_block2_1_bn (BatchNormalization)	(None, 7, 7, 512)	2,
conv5_block2_1_relu (Activation)	(None, 7, 7, 512)	
conv5_block2_2_conv (Conv2D)	(None, 7, 7, 512)	2,359,
conv5_block2_2_bn (BatchNormalization)	(None, 7, 7, 512)	2,
conv5_block2_2_relu (Activation)	(None, 7, 7, 512)	
conv5_block2_3_conv (Conv2D)	(None, 7, 7, 2048)	1,050,
conv5_block2_3_bn	(None, 7, 7, 2048)	8,

(BatchNormalization)		
conv5_block2_add (Add)	(None, 7, 7, 2048)	
conv5_block2_out (Activation)	(None, 7, 7, 2048)	
conv5_block3_1_conv (Conv2D)	(None, 7, 7, 512)	1,049,
conv5_block3_1_bn (BatchNormalization)	(None, 7, 7, 512)	2,
conv5_block3_1_relu (Activation)	(None, 7, 7, 512)	
conv5_block3_2_conv (Conv2D)	(None, 7, 7, 512)	2,359,
conv5_block3_2_bn (BatchNormalization)	(None, 7, 7, 512)	2,
conv5_block3_2_relu (Activation)	(None, 7, 7, 512)	
conv5_block3_3_conv (Conv2D)	(None, 7, 7, 2048)	1,050,
conv5_block3_3_bn (BatchNormalization)	(None, 7, 7, 2048)	8,
conv5_block3_add (Add)	(None, 7, 7, 2048)	
conv5_block3_out (Activation)	(None, 7, 7, 2048)	
avg_pool (GlobalAveragePooling2D)	(None, 2048)	
predictions (Dense)	(None, 1000)	2,049,

Total params: 25,636,712 (97.80 MB)

Trainable params: 25,583,592 (97.59 MB)

Non-trainable params: 53,120 (207.50 KB)

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
Downloading data from https://storage.googleapis.com/download.tensorflow.org/dat
a/imagenet_class_index.json
35363/35363 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
[('n02123045', 'tabby', 0.40373057)]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 104ms/step
[('n02106382', 'Bouvier_des_Flandres', 0.31675124)]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 105ms/step
[('n02124075', 'Egyptian_cat', 0.17904663)]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 106ms/step
[('n02124075', 'Egyptian_cat', 0.50363517)]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 102ms/step
[('n02123045', 'tabby', 0.75102234)]
```

Part 8 (OPTIONAL)

Set up `Ray Tune` and run automatic hyper parameter optimization for the CNN model as we have done in the DNN lab. Remember that you have to define the `train_CNN` function, specify the hyper parameter search space and the number of samples to evaluate, among other.