CNN_Lab_2023

April 29, 2024

Partner-1: Siddhesh Sreedar (sidsr770)

Partner-2: Hugo Morvan (hugmo418)

1 CNN Image Classification Laboration

Images used in this laboration are from CIFAR 10 (https://en.wikipedia.org/wiki/CIFAR-10). The CIFAR-10 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.

You need to answer all questions in this notebook.

1.1 Part 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.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve2d.html

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).

[ ]:
```
# This cell is finished

from scipy import signal
import numpy as np

# Get a test image
from scipy import misc
image = misc.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
```

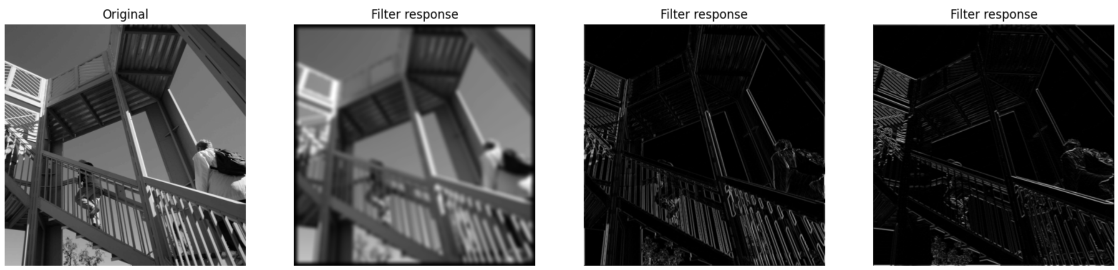
```
    fspecial('gaussian',[shape],[sigma])
    """
    m,n = [(ss-1.)/2. for ss in shape]
    y,x = np.ogrid[-m:m+1,-n:n+1]
    h = np.exp( -(x*x + y*y) / (2.*sigma*sigma) )
    h[ h < np.finfo(h.dtype).eps*h.max() ] = 0
    sumh = h.sum()
    if sumh != 0:
        h /= sumh
    return h

# Create Gaussian filter with certain size and standard deviation
gaussFilter = matlab_style_gauss2D((15,15),4)

# Define filter kernels for SobelX and Sobely
sobelX = np.array([[ 1, 0,  -1],
                    [2, 0, -2],
                    [1, 0, -1]])

sobelY = np.array([[ 1, 2,  1],
                    [0, 0, 0],
                    [-1, -2, -1]])
```

```
<ipython-input-1-f842718450cf>:8: DeprecationWarning: scipy.misc.ascent has been
deprecated in SciPy v1.10.0; and will be completely removed in SciPy v1.12.0.
Dataset methods have moved into the scipy.datasets module. Use
scipy.datasets.ascent instead.
  image = misc.ascent()
```

[ ]:
```
# Perform convolution using the function 'convolve2d' for the different filters
filterResponseGauss = signal.convolve2d(image,gaussFilter)
filterResponseSobelX = signal.convolve2d(image,sobelX)
filterResponseSobelY = signal.convolve2d(image,sobelY)
```

[ ]:
```
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('Filter response')
ax_filt1.set_axis_off()
ax_filt2.imshow(np.absolute(filterResponseSobelX), cmap='gray')
ax_filt2.set_title('Filter response')
```

```
ax_filt2.set_axis_off()
ax_filt3.imshow(np.absolute(filterResponseSobelY), cmap='gray')
ax_filt3.set_title('Filter response')
ax_filt3.set_axis_off()
```



1.2 Part 2: Understanding convolutions

Question 1: What do the 3 different filters (Gaussian, SobelX, SobelY) do to the original image?

Question 2: What is the size of the original image? How many channels does it have? How many channels does a color image normally have?

Question 3: What is the size of the different filters?

Question 4: What is the size of the filter response if mode 'same' is used for the convolution ?

Question 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?

Question 6: Why are 'valid' convolutions a problem for CNNs with many layers?

Question 1: The Gaussian filter is used to reduce the noise in the image and also blurs the image. The SobelX filter helps detect edges in the horizontal direction and the SobelY filter helps detect edges in the vertical direction.

Question 2: Size of the original image is 512 x 512. It has only one channel since its grayscale. For color, there is 3 channel (RGB)

Question 3: Gaussian -> 15 x 15, sobelX -> 3 x 3 , sobeY -> 3 X 3

Question 4: When mode = "same", the size for all 3 is 512 x 512

Question 5: When mode = "valid", the size of Gaussian -> 498 x 498, the size of sobelX -> 510 x 510, the size of sobelY -> 510 x 510. The size of the valid filter becomes (Image size) - (size of the filter - 1)

Question 6: Because there is information loss in each layer due to the filtering that happens in each layer. The size of the filter response reduces in each layer due to the valid mode, so there is information loss.

[ ]:
```
# Your code for checking sizes of image and filter responses
```

```
print(image.shape)

print(gaussFilter.shape)

print(sobelX.shape)

print(sobelY.shape)

filterResponseGauss_1 = signal.convolve2d(image,gaussFilter, mode = "valid")
print(filterResponseGauss_1.shape)

filterResponseSobelX = signal.convolve2d(image,sobelX, mode = "valid")
print(filterResponseSobelX.shape)
filterResponseSobelY = signal.convolve2d(image,sobelY, mode = "valid")
print(filterResponseSobelY.shape)
```

```
(512, 512)
(15, 15)
(3, 3)
(3, 3)
(498, 498)
(510, 510)
(510, 510)
```

1.3 Part 3: Get a graphics card

Skip this part if you run on a CPU (recommended)

Let's make sure that our script can see the graphics card that will be used. The graphics cards will perform all the time consuming convolutions in every training iteration.

[ ]:
```
import os
import warnings

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

import keras.backend as K
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";

# Allow growth of GPU memory, otherwise it will always look like all the memory␣
↪is being used
physical_devices = tf.config.experimental.list_physical_devices('GPU')
```

```
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

1.4 Part 4: How fast is the graphics card?

Question 7: Why are the filters used for a color image of size 7 x 7 x 3, and not 7 x 7 ?

Question 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?

Question 9: 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.

Question 7: This is beacuse color images have 3 channels (RGB) so we need one fillter for each of the channels.

Question 8: It produces a different result compared to signal.convolve2d . In "signal.convolve2d", the filter (Kernel) is inverted before applying to the image while in "Conv2D" it is not.

Question 9: For a batch of 3 images, using a graphic card compared to the CPU might not yield drastic change in computational efficiency but for 1,000 images we should notice computational efficiency with using graphic card as comapred to the CPU. Graphic cards are designed to do multiple task in parallel making it perform overall tasks faster than the CPU.

1.5 Part 5: Load data

Time to make a 2D CNN. Load the images and labels from keras.datasets, this cell is already finished.

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

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']

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

print("Training images have size {} and labels have size {} ".format(Xtrain.
 ↪shape, Ytrain.shape))
print("Test images have size {} and labels have size {} \n ".format(Xtest.
 ↪shape, Ytest.shape))

# Reduce the number of images for training and testing to 10000 and 2000␣
 ↪respectively,
# to reduce processing time for this laboration
Xtrain = Xtrain[0:10000]
Ytrain = Ytrain[0:10000]

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

```
Ytestint = Ytest

print("Reduced training images have size %s and labels have size %s " % (Xtrain.
↪shape, Ytrain.shape))
print("Reduced test images have size %s and labels have size %s \n" % (Xtest.
↪shape, Ytest.shape))

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

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 13s 0us/step
Training images have size (50000, 32, 32, 3) and labels have size (50000, 1)
Test images have size (10000, 32, 32, 3) and labels have size (10000, 1)

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

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

1.6 Part 6: Plotting

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

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

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

    plt.subplot(3,6,i+1)
    plt.tight_layout()
```

```
    plt.imshow(Xtrain[idx])
    plt.title("Class: {} ({})".format(label, classes[label]))
    plt.axis('off')
plt.show()
```

1.7 Part 7: Split data into training, validation and testing

Split your training data into training (Xtrain, Ytrain) and validation (Xval, Yval), so that we have training, validation and test datasets (as in the previous laboration). We use a function in scikit learn. Use 25% of the data for validation.

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

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

# Your code for splitting the dataset
Xtrain, Xval, Ytrain, Yval = train_test_split(Xtrain, Ytrain, test_size=0.25,␣
 ↪random_state=123)


# Print the size of training data, validation data and test data
print('Xtrain has size {}.'.format(Xtrain.shape))
print('Ytrain has size {}.'.format(Ytrain.shape))

print('Xval has size {}.'.format(Xval.shape))
print('Yval has size {}.'.format(Yval.shape))

print('Xtest has size {}.'.format(Xtest.shape))
print('Ytest has size {}.'.format(Ytest.shape))
```

```
Xtrain has size (7500, 32, 32, 3).
Ytrain has size (7500, 1).
Xval has size (2500, 32, 32, 3).
Yval has size (2500, 1).
```

```
Xtest has size (2000, 32, 32, 3).
Ytest has size (2000, 1).
```

1.8 Part 8: Preprocessing of images

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. This cell is already finished.

[ ]:
```python
# 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
```

1.9 Part 9: Preprocessing of labels

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 a function in Keras, see https://keras.io/api/utils/python_utils/#to_categorical-function

[ ]:
```python
from tensorflow.keras.utils import to_categorical

# Print shapes before converting the labels
print(Ytrain.shape)
print(Yval.shape)
print(Ytest.shape)


# Your code for converting Ytrain, Yval, Ytest to categorical
Ytrain_enc=to_categorical((Ytrain),10)
Yval_enc=to_categorical((Yval),10)
Ytest_enc=to_categorical((Ytest),10)


# Print shapes after converting the labels
print(Ytrain_enc.shape)
print(Yval_enc.shape)
print(Ytest_enc.shape)
```

```
(7500, 1)
(2500, 1)
(2000, 1)
(7500, 10)
```

```
(2500, 10)
(2000, 10)
```

1.10 Part 10: 2D CNN

Finish this code to create the image classifier, using a 2D CNN. Each convolutional layer will contain 2D convolution, batch normalization and max pooling. After the convolutional layers comes a flatten layer and a number of intermediate dense layers. The convolutional layers should take the number of filters as an argument, use a kernel size of 3 x 3, 'same' padding, and relu activation functions. The number of filters will double with each convolutional layer. The max pooling layers should have a pool size of 2 x 2. The intermediate dense layers before the final dense layer should take the number of nodes as an argument, use relu activation functions, and be followed by batch normalization. The final dense layer should have 10 nodes (= the number of classes in this laboration) and 'softmax' activation. Here we start with the Adam optimizer.

Relevant functions are

`model.add()`, adds a layer to the network

`Dense()`, a dense network layer

`Conv2D()`, performs 2D convolutions with a number of filters with a certain size (e.g. 3 x 3).

`BatchNormalization()`, perform batch normalization

`MaxPooling2D()`, saves the max for a given pool size, results in down sampling

`Flatten()`, flatten a multi-channel tensor into a long vector

`model.compile()`, compile the model, add " metrics=['accuracy'] " to print the classification accuracy during the training

See https://keras.io/api/layers/core_layers/dense/ and https://keras.io/api/layers/reshaping_layers/flatten/ for information on how the `Dense()` and `Flatten()` functions work

See https://keras.io/layers/convolutional/ for information on how `Conv2D()` works

See https://keras.io/layers/pooling/ for information on how `MaxPooling2D()` works

Import a relevant cost function for multi-class classification from keras.losses (https://keras.io/losses/) , it relates to how many classes you have.

See the following links for how to compile, train and evaluate the model

https://keras.io/api/models/model_training_apis/#compile-method

https://keras.io/api/models/model_training_apis/#fit-method

https://keras.io/api/models/model_training_apis/#evaluate-method

[ ]:
```
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D, BatchNormalization, MaxPooling2D,
    Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from keras.losses import CategoricalCrossentropy
```

```
# Set seed from random number generator, for better comparisons
from numpy.random import seed
seed(123)

def build_CNN(input_shape, n_conv_layers=2, n_filters=16, n_dense_layers=0,␣
↪n_nodes=50, use_dropout=False, learning_rate=0.01):

    # Setup a sequential model
    model = Sequential()

    # Add first convolutional layer to the model, requires input shape
    model.add(Conv2D(filters=n_filters,kernel_size=(3,3), padding = "same",
        activation= "ReLU" , input_shape=input_shape ))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Add remaining convolutional layers to the model, the number of filters␣
↪should increase a factor 2 for each layer
    for i in range(n_conv_layers-1):
      n_filters = n_filters*2
      model.add(Conv2D(filters=n_filters,kernel_size=(3,3), padding = "same",
        activation= "ReLU"))
      model.add(MaxPooling2D(pool_size=(2, 2)))


    # Add flatten layer
    model.add(Flatten())


    # Add intermediate dense layers
    for i in range(n_dense_layers):
      model.add(Dense(n_nodes, activation = "ReLU"))
      model.add(BatchNormalization())
      if use_dropout:
        model.add(Dropout(rate=0.5))

    # Add final dense layer
    model.add(Dense(10 , activation="softmax"))

    # Compile model
    model.compile(loss = CategoricalCrossentropy(), metrics=["accuracy"],␣
↪optimizer = Adam(learning_rate=learning_rate))
    #model.summary()


    return model
```

[ ]:
```
# Lets define a help function for plotting the training results
import matplotlib.pyplot as plt
def plot_results(history):

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

    plt.figure(figsize=(10,4))
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.plot(loss)
    plt.plot(val_loss)
    plt.legend(['Training','Validation'])

    plt.figure(figsize=(10,4))
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.plot(acc)
    plt.plot(val_acc)
    plt.legend(['Training','Validation'])

    plt.show()
```

1.11 Part 11: Train 2D CNN

Time to train the 2D CNN, start with 2 convolutional layers, no intermediate dense layers, learning rate = 0.01. The first convolutional layer should have 16 filters (which means that the second convolutional layer will have 32 filters).

Relevant functions

`build_CNN`, the function we defined in Part 10, call it with the parameters you want to use

`model.fit()`, train the model with some training data

`model.evaluate()`, apply the trained model to some test data

See the following links for how to train and evaluate the model

https://keras.io/api/models/model_training_apis/#fit-method

https://keras.io/api/models/model_training_apis/#evaluate-method

1.12 2 convolutional layers, no intermediate dense layers

[ ]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = (32,32,3)
```

```
# Build model
model1 = build_CNN(input_shape, n_conv_layers=2, n_filters=16,learning_rate=0.
 ↪01)

# Train the model  using training data and validation data
history1 = model1.fit(Xtrain, Ytrain_enc, validation_data=(Xval, Yval_enc),␣
 ↪epochs=epochs, batch_size=batch_size)
```

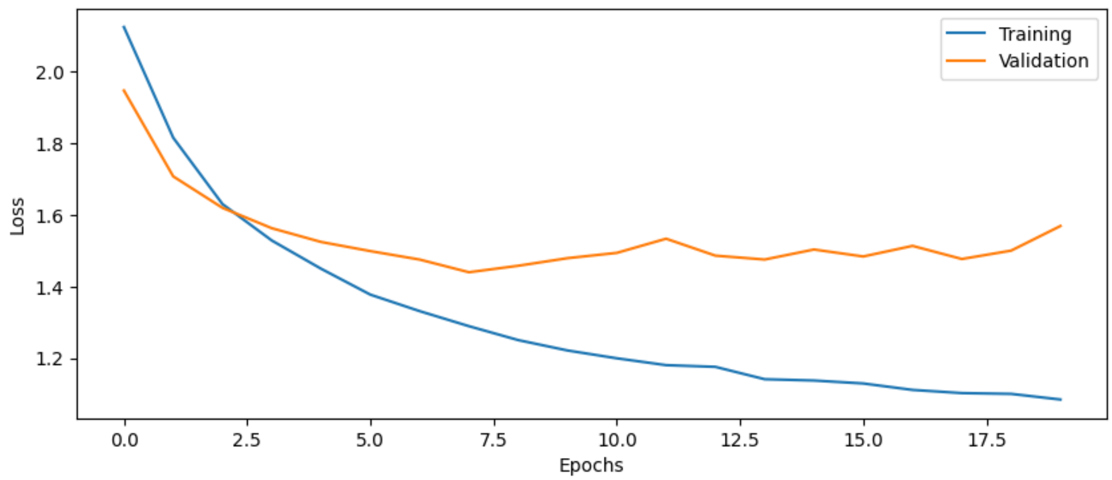
```
Epoch 1/20
75/75 [==============================] - 5s 9ms/step - loss: 2.1244 - accuracy:
0.2308 - val_loss: 1.9472 - val_accuracy: 0.2832
Epoch 2/20
75/75 [==============================] - 0s 5ms/step - loss: 1.8154 - accuracy:
0.3481 - val_loss: 1.7075 - val_accuracy: 0.3872
Epoch 3/20
75/75 [==============================] - 0s 5ms/step - loss: 1.6301 - accuracy:
0.4156 - val_loss: 1.6196 - val_accuracy: 0.4124
Epoch 4/20
75/75 [==============================] - 0s 5ms/step - loss: 1.5285 - accuracy:
0.4544 - val_loss: 1.5627 - val_accuracy: 0.4464
Epoch 5/20
75/75 [==============================] - 0s 5ms/step - loss: 1.4496 - accuracy:
0.4800 - val_loss: 1.5244 - val_accuracy: 0.4612
Epoch 6/20
75/75 [==============================] - 0s 5ms/step - loss: 1.3773 - accuracy:
0.5093 - val_loss: 1.4988 - val_accuracy: 0.4804
Epoch 7/20
75/75 [==============================] - 0s 5ms/step - loss: 1.3313 - accuracy:
0.5247 - val_loss: 1.4752 - val_accuracy: 0.4776
Epoch 8/20
75/75 [==============================] - 0s 5ms/step - loss: 1.2892 - accuracy:
0.5411 - val_loss: 1.4398 - val_accuracy: 0.4960
Epoch 9/20
75/75 [==============================] - 0s 5ms/step - loss: 1.2503 - accuracy:
0.5585 - val_loss: 1.4579 - val_accuracy: 0.5020
Epoch 10/20
75/75 [==============================] - 0s 5ms/step - loss: 1.2214 - accuracy:
0.5655 - val_loss: 1.4791 - val_accuracy: 0.5048
Epoch 11/20
75/75 [==============================] - 0s 5ms/step - loss: 1.1995 - accuracy:
0.5797 - val_loss: 1.4939 - val_accuracy: 0.4952
Epoch 12/20
75/75 [==============================] - 0s 5ms/step - loss: 1.1804 - accuracy:
0.5843 - val_loss: 1.5334 - val_accuracy: 0.4800
Epoch 13/20
75/75 [==============================] - 0s 5ms/step - loss: 1.1757 - accuracy:
```

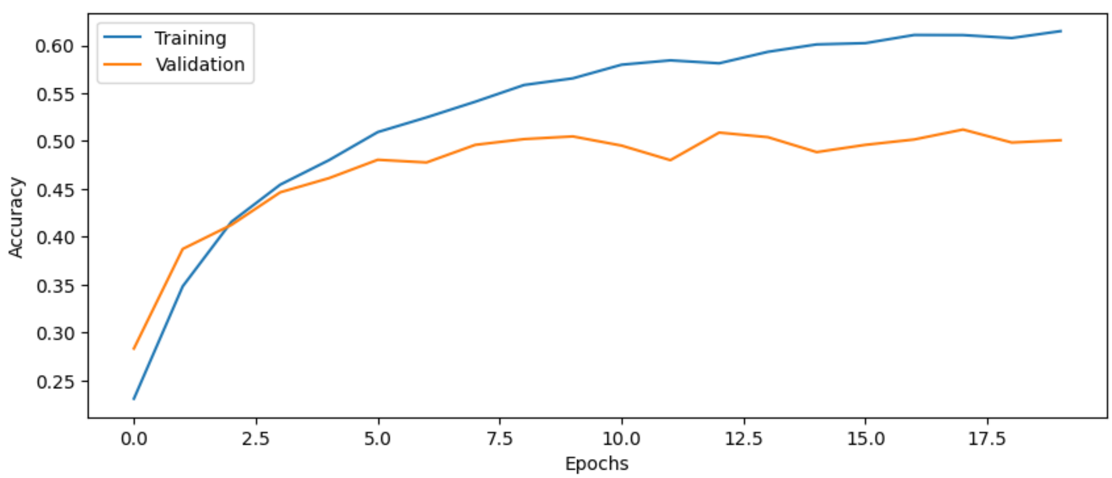
```
0.5812 - val_loss: 1.4862 - val_accuracy: 0.5088
Epoch 14/20
75/75 [==============================] - 0s 6ms/step - loss: 1.1412 - accuracy:
0.5932 - val_loss: 1.4753 - val_accuracy: 0.5040
Epoch 15/20
75/75 [==============================] - 0s 5ms/step - loss: 1.1375 - accuracy:
0.6009 - val_loss: 1.5033 - val_accuracy: 0.4884
Epoch 16/20
75/75 [==============================] - 0s 5ms/step - loss: 1.1293 - accuracy:
0.6023 - val_loss: 1.4838 - val_accuracy: 0.4960
Epoch 17/20
75/75 [==============================] - 0s 5ms/step - loss: 1.1113 - accuracy:
0.6108 - val_loss: 1.5134 - val_accuracy: 0.5016
Epoch 18/20
75/75 [==============================] - 0s 6ms/step - loss: 1.1024 - accuracy:
0.6107 - val_loss: 1.4768 - val_accuracy: 0.5120
Epoch 19/20
75/75 [==============================] - 1s 7ms/step - loss: 1.1002 - accuracy:
0.6076 - val_loss: 1.5002 - val_accuracy: 0.4984
Epoch 20/20
75/75 [==============================] - 1s 8ms/step - loss: 1.0845 - accuracy:
0.6148 - val_loss: 1.5689 - val_accuracy: 0.5008
```

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

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

```
20/20 [==============================] - 0s 3ms/step - loss: 1.5552 - accuracy:
0.4960
Test loss: 1.5552
Test accuracy: 0.4960
```

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





1.13 Part 12: Improving performance

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

Question 10: How big is the difference between training and test accuracy?

Question 11: For the DNN laboration we used a batch size of 10,000, why do we need to use a smaller batch size in this laboration?

Consisdering that we created a simple CNN for 10 classes, we get 50% accuracy which is better than a random chance which is 10%

Question 10: Test accuracy: 0.5025 and training accuracy: 0.6807

Question 11: Having a very high batch size will more computational intensive for a CNN model. Additionally, the dataset was more for the DNN lab compared to this lab.

1.14 2 convolutional layers, 1 intermediate dense layer (50 nodes)

[ ]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = (32,32,3)

# Build model
model2 = build_CNN(input_shape, n_conv_layers=2, n_filters=16,␣
 ↪n_dense_layers=1, n_nodes=50, learning_rate=0.01)

# Train the model  using training data and validation data
history2 =  model2.fit(Xtrain, Ytrain_enc, validation_data=(Xval, Yval_enc),␣
 ↪epochs=epochs, batch_size=batch_size)
```

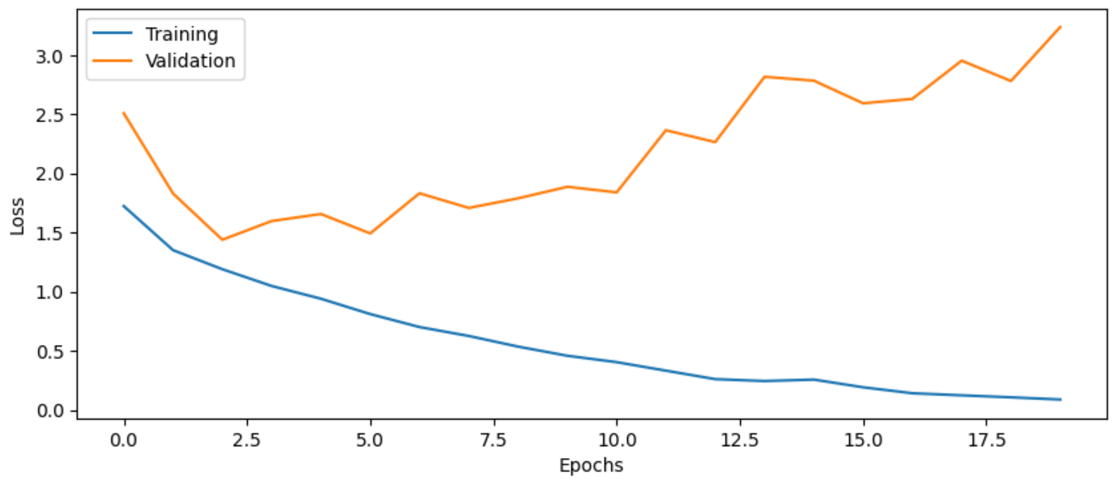
```
Epoch 1/20
75/75 [==============================] - 4s 9ms/step - loss: 1.7231 - accuracy:
0.3867 - val_loss: 2.5080 - val_accuracy: 0.3528
Epoch 2/20
75/75 [==============================] - 0s 6ms/step - loss: 1.3508 - accuracy:
0.5135 - val_loss: 1.8296 - val_accuracy: 0.4300
Epoch 3/20
75/75 [==============================] - 0s 6ms/step - loss: 1.1897 - accuracy:
0.5764 - val_loss: 1.4389 - val_accuracy: 0.5232
Epoch 4/20
75/75 [==============================] - 0s 6ms/step - loss: 1.0481 - accuracy:
0.6247 - val_loss: 1.5968 - val_accuracy: 0.5012
Epoch 5/20
75/75 [==============================] - 0s 6ms/step - loss: 0.9401 - accuracy:
0.6680 - val_loss: 1.6565 - val_accuracy: 0.5212
Epoch 6/20
75/75 [==============================] - 0s 6ms/step - loss: 0.8105 - accuracy:
0.7100 - val_loss: 1.4924 - val_accuracy: 0.5500
Epoch 7/20
75/75 [==============================] - 0s 6ms/step - loss: 0.7008 - accuracy:
0.7517 - val_loss: 1.8315 - val_accuracy: 0.5172
Epoch 8/20
75/75 [==============================] - 0s 6ms/step - loss: 0.6254 - accuracy:
0.7747 - val_loss: 1.7077 - val_accuracy: 0.5492
Epoch 9/20
75/75 [==============================] - 0s 5ms/step - loss: 0.5354 - accuracy:
0.8073 - val_loss: 1.7888 - val_accuracy: 0.5680
Epoch 10/20
75/75 [==============================] - 1s 9ms/step - loss: 0.4580 - accuracy:
0.8355 - val_loss: 1.8865 - val_accuracy: 0.5392
Epoch 11/20
75/75 [==============================] - 1s 11ms/step - loss: 0.4042 - accuracy:
0.8577 - val_loss: 1.8394 - val_accuracy: 0.5560
```

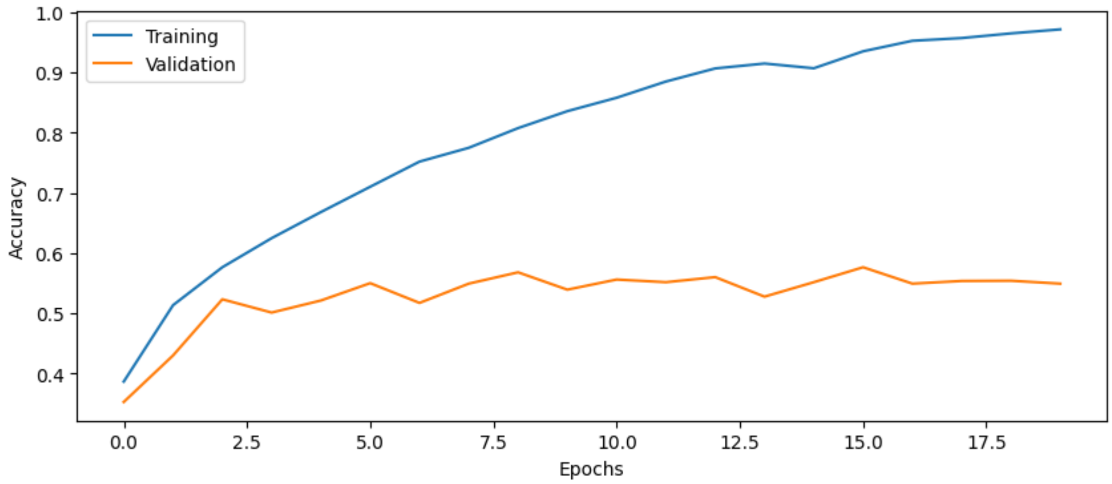
```
Epoch 12/20
75/75 [==============================] - 1s 10ms/step - loss: 0.3321 - accuracy:
0.8847 - val_loss: 2.3646 - val_accuracy: 0.5516
Epoch 13/20
75/75 [==============================] - 1s 7ms/step - loss: 0.2606 - accuracy:
0.9064 - val_loss: 2.2644 - val_accuracy: 0.5600
Epoch 14/20
75/75 [==============================] - 0s 6ms/step - loss: 0.2444 - accuracy:
0.9145 - val_loss: 2.8163 - val_accuracy: 0.5276
Epoch 15/20
75/75 [==============================] - 0s 6ms/step - loss: 0.2567 - accuracy:
0.9067 - val_loss: 2.7839 - val_accuracy: 0.5516
Epoch 16/20
75/75 [==============================] - 0s 6ms/step - loss: 0.1915 - accuracy:
0.9348 - val_loss: 2.5925 - val_accuracy: 0.5764
Epoch 17/20
75/75 [==============================] - 0s 6ms/step - loss: 0.1411 - accuracy:
0.9523 - val_loss: 2.6296 - val_accuracy: 0.5492
Epoch 18/20
75/75 [==============================] - 1s 7ms/step - loss: 0.1236 - accuracy:
0.9568 - val_loss: 2.9528 - val_accuracy: 0.5536
Epoch 19/20
75/75 [==============================] - 1s 7ms/step - loss: 0.1067 - accuracy:
0.9645 - val_loss: 2.7806 - val_accuracy: 0.5540
Epoch 20/20
75/75 [==============================] - 1s 8ms/step - loss: 0.0882 - accuracy:
0.9712 - val_loss: 3.2369 - val_accuracy: 0.5492
```

[ ]:
```python
# Evaluate the trained model on test set, not used in training or validation
score = model2.evaluate(Xtest, Ytest_enc, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
20/20 [==============================] - 0s 3ms/step - loss: 2.9601 - accuracy:
0.5480
Test loss: 2.9601
Test accuracy: 0.5480
```

[ ]:
```python
# Plot the history from the training run
plot_results(history2)
```





1.15 4 convolutional layers, 1 intermediate dense layer (50 nodes)

[ ]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = (32,32,3)

# Build model
model3 = build_CNN(input_shape, n_conv_layers=4, n_filters=16,
    n_dense_layers=1, n_nodes=50, learning_rate=0.01)

# Train the model  using training data and validation data
```

```python
history3 = model3.fit(Xtrain, Ytrain_enc, validation_data=(Xval, Yval_enc), epochs=epochs, batch_size=batch_size)
```

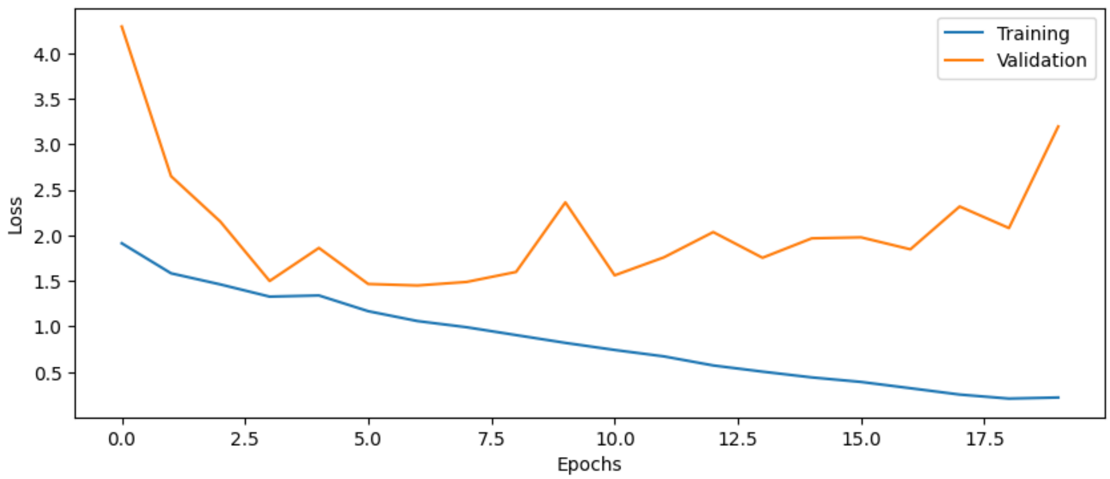
```
Epoch 1/20
75/75 [==============================] - 4s 14ms/step - loss: 1.9141 - accuracy:
0.2875 - val_loss: 4.2951 - val_accuracy: 0.1768
Epoch 2/20
75/75 [==============================] - 1s 10ms/step - loss: 1.5844 - accuracy:
0.4163 - val_loss: 2.6513 - val_accuracy: 0.2652
Epoch 3/20
75/75 [==============================] - 1s 16ms/step - loss: 1.4623 - accuracy:
0.4587 - val_loss: 2.1522 - val_accuracy: 0.3536
Epoch 4/20
75/75 [==============================] - 1s 11ms/step - loss: 1.3282 - accuracy:
0.5156 - val_loss: 1.4999 - val_accuracy: 0.4852
Epoch 5/20
75/75 [==============================] - 1s 7ms/step - loss: 1.3410 - accuracy:
0.5079 - val_loss: 1.8630 - val_accuracy: 0.4132
Epoch 6/20
75/75 [==============================] - 1s 7ms/step - loss: 1.1683 - accuracy:
0.5728 - val_loss: 1.4670 - val_accuracy: 0.4952
Epoch 7/20
75/75 [==============================] - 0s 7ms/step - loss: 1.0598 - accuracy:
0.6184 - val_loss: 1.4506 - val_accuracy: 0.5192
Epoch 8/20
75/75 [==============================] - 0s 6ms/step - loss: 0.9918 - accuracy:
0.6401 - val_loss: 1.4897 - val_accuracy: 0.5112
Epoch 9/20
75/75 [==============================] - 0s 6ms/step - loss: 0.9058 - accuracy:
0.6713 - val_loss: 1.5992 - val_accuracy: 0.5112
Epoch 10/20
75/75 [==============================] - 0s 6ms/step - loss: 0.8202 - accuracy:
0.7035 - val_loss: 2.3633 - val_accuracy: 0.4348
Epoch 11/20
75/75 [==============================] - 0s 6ms/step - loss: 0.7423 - accuracy:
0.7349 - val_loss: 1.5621 - val_accuracy: 0.5488
Epoch 12/20
75/75 [==============================] - 0s 6ms/step - loss: 0.6717 - accuracy:
0.7585 - val_loss: 1.7596 - val_accuracy: 0.5536
Epoch 13/20
75/75 [==============================] - 0s 6ms/step - loss: 0.5722 - accuracy:
0.7964 - val_loss: 2.0371 - val_accuracy: 0.5024
Epoch 14/20
75/75 [==============================] - 1s 7ms/step - loss: 0.5042 - accuracy:
0.8173 - val_loss: 1.7549 - val_accuracy: 0.5744
Epoch 15/20
75/75 [==============================] - 1s 8ms/step - loss: 0.4408 - accuracy:
```

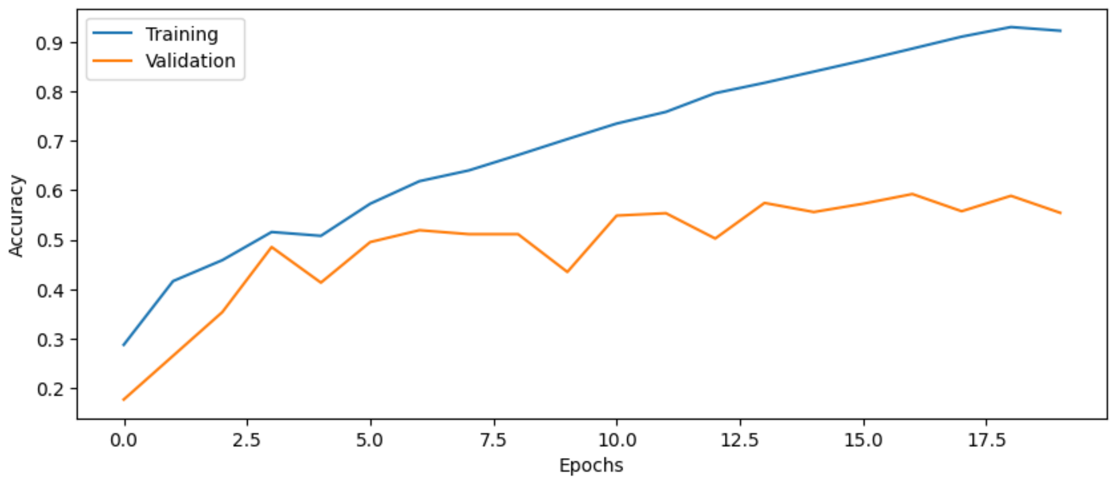
```
0.8400 - val_loss: 1.9686 - val_accuracy: 0.5560
Epoch 16/20
75/75 [==============================] - 1s 8ms/step - loss: 0.3911 - accuracy:
0.8627 - val_loss: 1.9794 - val_accuracy: 0.5728
Epoch 17/20
75/75 [==============================] - 0s 7ms/step - loss: 0.3221 - accuracy:
0.8865 - val_loss: 1.8477 - val_accuracy: 0.5924
Epoch 18/20
75/75 [==============================] - 0s 6ms/step - loss: 0.2524 - accuracy:
0.9107 - val_loss: 2.3182 - val_accuracy: 0.5576
Epoch 19/20
75/75 [==============================] - 1s 8ms/step - loss: 0.2081 - accuracy:
0.9301 - val_loss: 2.0814 - val_accuracy: 0.5888
Epoch 20/20
75/75 [==============================] - 0s 6ms/step - loss: 0.2201 - accuracy:
0.9227 - val_loss: 3.1974 - val_accuracy: 0.5544
```

[ ]:
```python
# Evaluate the trained model on test set, not used in training or validation
score = model3.evaluate(Xtest, Ytest_enc, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
20/20 [==============================] - 0s 5ms/step - loss: 3.1183 - accuracy:
0.5350
Test loss: 3.1183
Test accuracy: 0.5350
```

[ ]:
```python
# Plot the history from the training run
plot_results(history3)
```





1.16 Part 13: Plot the CNN architecture

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

Question 12: How many trainable parameters does your network have? Which part of the network contains most of the parameters?

Question 13: What is the input to and output of a Conv2D layer? What are the dimensions of the input and output?

Question 14: Is the batch size always the first dimension of each 4D tensor? Check the documentation for Conv2D, https://keras.io/layers/convolutional/

Question 15: 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?

Question 16: 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)?

Question 17: How does MaxPooling help in reducing the number of parameters to train?

Question 12: Total params: 123800 . The 4th Convolutional Layer has the most number of parameters (73856)

Question 13: For the first layer, the output is (batch_size, 32,32,3) and once it passes through the first convolutional layer, it beconmes (batch_size,32,32,16), it changes to 16 due to the 16 filters that is applied to the image.

Input dim: (batch_size,32,32,3) Ouptput dim: (batch_size,10)

Question 14: Yes, it is always the first dimension.

Question 15: 128 channels

Question 16: Cause we also need to add the different channels as well which in this case is 3 (RGB) since each of the filter is applied to those 3 channels as well thus creating additional parameters.

Question 17: Max pooling takes the maximum value within a local neighborhood of pixels. There by capturing dominant features in the input while also reducing the shape it outputs due to this.

[ ]:
```
# Print network architecture

model3.summary()
```

```
Model: "sequential_2"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 32, 32, 16)        448

 max_pooling2d_4 (MaxPoolin  (None, 16, 16, 16)        0
 g2D)

 conv2d_5 (Conv2D)           (None, 16, 16, 32)        4640

 max_pooling2d_5 (MaxPoolin  (None, 8, 8, 32)          0
 g2D)

 conv2d_6 (Conv2D)           (None, 8, 8, 64)          18496

 max_pooling2d_6 (MaxPoolin  (None, 4, 4, 64)          0
 g2D)

 conv2d_7 (Conv2D)           (None, 4, 4, 128)         73856

 max_pooling2d_7 (MaxPoolin  (None, 2, 2, 128)         0
 g2D)

 flatten_2 (Flatten)         (None, 512)               0

 dense_3 (Dense)             (None, 50)                25650

 batch_normalization_1 (Bat  (None, 50)                200
 chNormalization)

 dense_4 (Dense)             (None, 10)                510

=================================================================
Total params: 123800 (483.59 KB)
Trainable params: 123700 (483.20 KB)
Non-trainable params: 100 (400.00 Byte)
_________________________________________________________________
```

1.17 Part 14: Dropout regularization

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

Question 18: How much did the test accuracy improve with dropout, compared to without dropout?

Question 19: What other types of regularization can be applied? How can you add L2 regularization for the convolutional layers?

Question 18: The test accuracy actually reduced from 0.5625 to 0.5225 when the dropout was used. This could be that the droput rate is too high and its removing a lot of important aspects from the model.

Question 19: Other types of regularization are: L1, L2 regularization , data augmentation, early stopping, bagging method.

We can do L2 regularization by adding this to each of the layers: kernel_regularizer=keras.regularizers.L2(lambda), keras.activity_regularizer=regularizers.L2(lambda))

1.18 4 convolutional layers, 1 intermediate dense layer (50 nodes), dropout

[ ]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = (32,32,3)

# Build model
model4 = build_CNN(input_shape, n_conv_layers=4, n_filters=16,␣
↪n_dense_layers=1, n_nodes=50, learning_rate=0.01, use_dropout = True )

# Train the model  using training data and validation data
history4 = model4.fit(Xtrain, Ytrain_enc, validation_data=(Xval, Yval_enc),␣
↪epochs=epochs, batch_size=batch_size)
```

```
Epoch 1/20
75/75 [==============================] - 4s 12ms/step - loss: 2.0314 - accuracy:
0.2511 - val_loss: 6.3296 - val_accuracy: 0.1068
Epoch 2/20
75/75 [==============================] - 1s 7ms/step - loss: 1.7332 - accuracy:
0.3407 - val_loss: 2.6524 - val_accuracy: 0.2552
Epoch 3/20
75/75 [==============================] - 1s 7ms/step - loss: 1.6477 - accuracy:
0.3769 - val_loss: 1.8586 - val_accuracy: 0.3996
Epoch 4/20
75/75 [==============================] - 1s 7ms/step - loss: 1.5630 - accuracy:
0.4227 - val_loss: 1.6150 - val_accuracy: 0.4020
Epoch 5/20
75/75 [==============================] - 1s 7ms/step - loss: 1.4982 - accuracy:
0.4444 - val_loss: 2.0963 - val_accuracy: 0.3552
Epoch 6/20
```

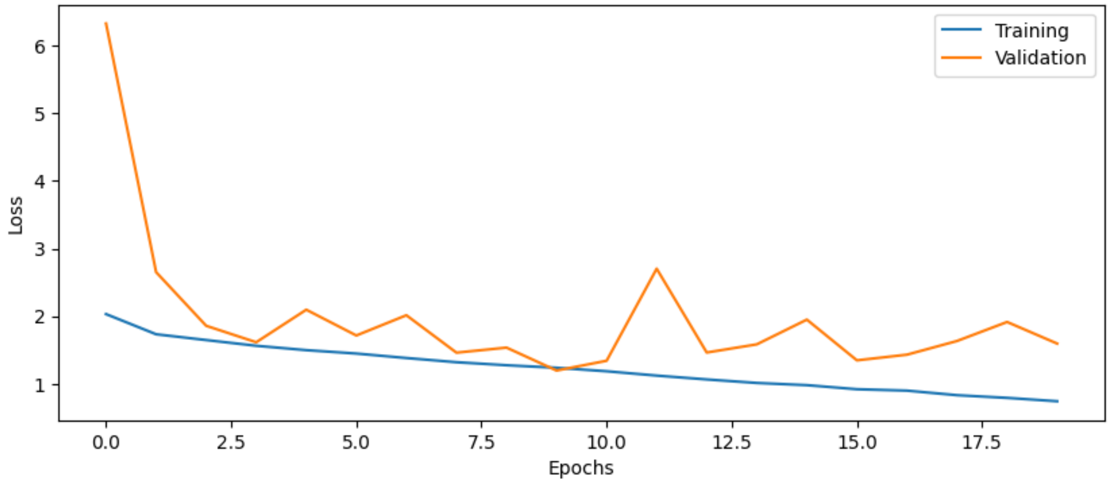
```
75/75 [==============================] - 0s 7ms/step - loss: 1.4480 - accuracy:
0.4627 - val_loss: 1.7151 - val_accuracy: 0.4144
Epoch 7/20
75/75 [==============================] - 0s 7ms/step - loss: 1.3818 - accuracy:
0.4829 - val_loss: 2.0145 - val_accuracy: 0.4128
Epoch 8/20
75/75 [==============================] - 1s 7ms/step - loss: 1.3197 - accuracy:
0.5152 - val_loss: 1.4610 - val_accuracy: 0.4544
Epoch 9/20
75/75 [==============================] - 1s 7ms/step - loss: 1.2752 - accuracy:
0.5329 - val_loss: 1.5356 - val_accuracy: 0.4632
Epoch 10/20
75/75 [==============================] - 0s 6ms/step - loss: 1.2373 - accuracy:
0.5459 - val_loss: 1.1965 - val_accuracy: 0.5796
Epoch 11/20
75/75 [==============================] - 1s 7ms/step - loss: 1.1860 - accuracy:
0.5743 - val_loss: 1.3414 - val_accuracy: 0.5312
Epoch 12/20
75/75 [==============================] - 0s 7ms/step - loss: 1.1218 - accuracy:
0.5879 - val_loss: 2.7028 - val_accuracy: 0.4080
Epoch 13/20
75/75 [==============================] - 1s 7ms/step - loss: 1.0650 - accuracy:
0.6155 - val_loss: 1.4631 - val_accuracy: 0.5412
Epoch 14/20
75/75 [==============================] - 0s 7ms/step - loss: 1.0129 - accuracy:
0.6287 - val_loss: 1.5840 - val_accuracy: 0.5188
Epoch 15/20
75/75 [==============================] - 1s 10ms/step - loss: 0.9804 - accuracy:
0.6439 - val_loss: 1.9498 - val_accuracy: 0.4852
Epoch 16/20
75/75 [==============================] - 1s 10ms/step - loss: 0.9203 - accuracy:
0.6605 - val_loss: 1.3473 - val_accuracy: 0.5656
Epoch 17/20
75/75 [==============================] - 1s 10ms/step - loss: 0.8994 - accuracy:
0.6728 - val_loss: 1.4314 - val_accuracy: 0.5580
Epoch 18/20
75/75 [==============================] - 1s 8ms/step - loss: 0.8319 - accuracy:
0.6937 - val_loss: 1.6338 - val_accuracy: 0.5316
Epoch 19/20
75/75 [==============================] - 0s 6ms/step - loss: 0.7924 - accuracy:
0.7047 - val_loss: 1.9150 - val_accuracy: 0.5144
Epoch 20/20
75/75 [==============================] - 1s 7ms/step - loss: 0.7426 - accuracy:
0.7316 - val_loss: 1.5956 - val_accuracy: 0.5508
```

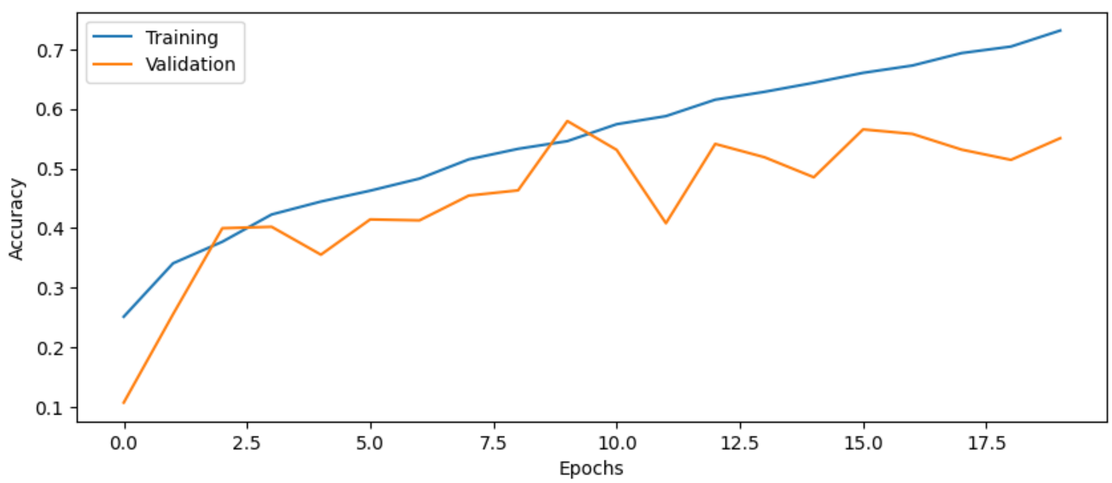
[ ]:
```
# Evaluate the trained model on test set, not used in training or validation
score = model4.evaluate(Xtest, Ytest_enc, batch_size=batch_size)
```

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

```
20/20 [==============================] - 0s 3ms/step - loss: 1.6627 - accuracy:
0.5620
Test loss: 1.6627
Test accuracy: 0.5620
```

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





1.19 Part 15: Tweaking 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.

Question 20: How high test accuracy can you obtain? What is your best configuration?

Question 20: The highest test accuray: 0.5990

Configuration: Batch_size: 100

epoch = 20

Conv_layers = 4

Kernel_filter = 32

dense layers + nodes = 1 , 50 nodes

learning rate = 0.01

dropout = False

1.20 Your best config

[ ]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = (32,32,3)


# Build model
model5 = build_CNN(input_shape, n_conv_layers=4, n_filters=32,␣
 ↪n_dense_layers=1, n_nodes=50, learning_rate=0.01 )

# Train the model  using training data and validation data
history5 = model5.fit(Xtrain, Ytrain_enc, validation_data=(Xval, Yval_enc),␣
 ↪epochs=epochs, batch_size=batch_size)
```

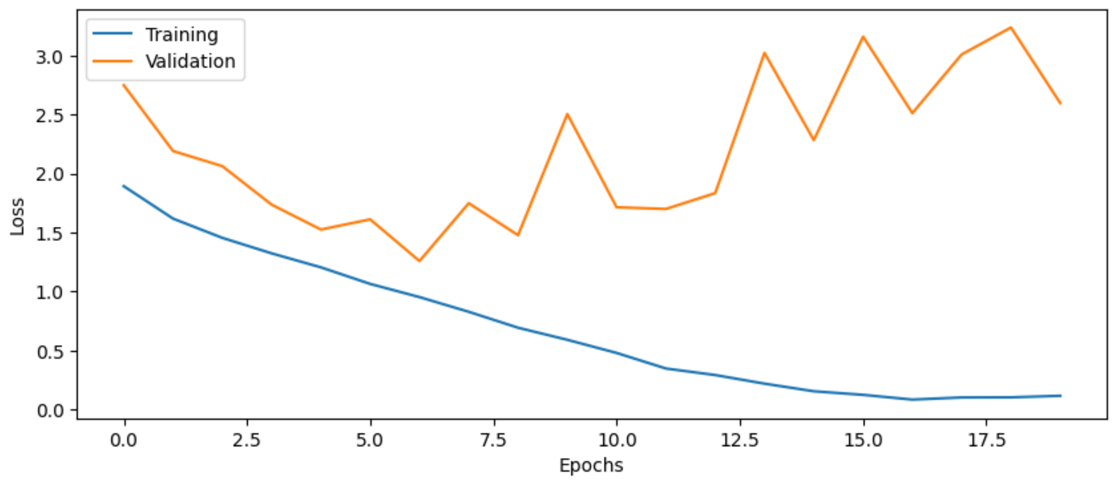
```
Epoch 1/20
75/75 [==============================] - 4s 11ms/step - loss: 1.8927 - accuracy:
0.2884 - val_loss: 2.7482 - val_accuracy: 0.2320
Epoch 2/20
75/75 [==============================] - 1s 8ms/step - loss: 1.6178 - accuracy:
0.3988 - val_loss: 2.1905 - val_accuracy: 0.3440
Epoch 3/20
75/75 [==============================] - 1s 8ms/step - loss: 1.4547 - accuracy:
0.4609 - val_loss: 2.0624 - val_accuracy: 0.3708
Epoch 4/20
75/75 [==============================] - 1s 7ms/step - loss: 1.3234 - accuracy:
0.5189 - val_loss: 1.7363 - val_accuracy: 0.4116
```

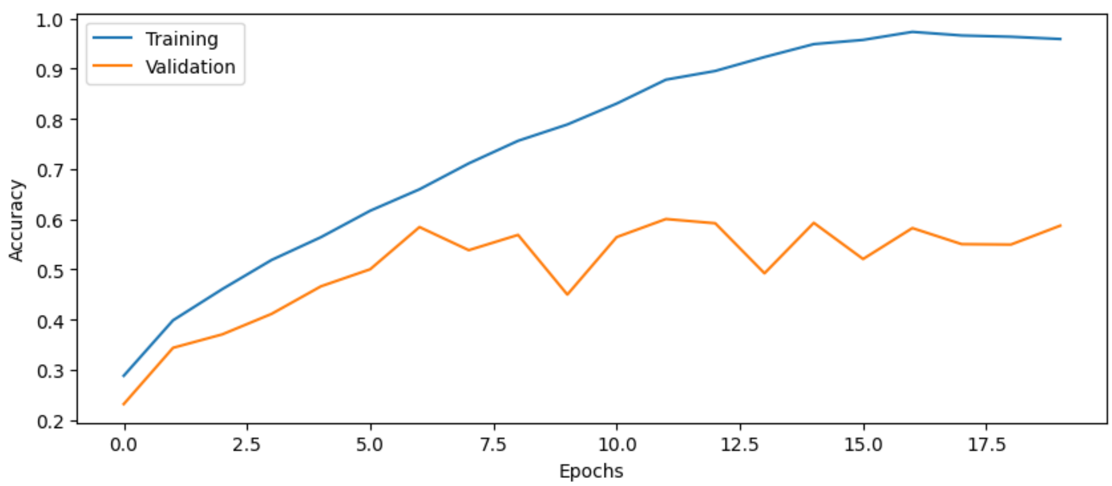
```
Epoch 5/20
75/75 [==============================] - 1s 7ms/step - loss: 1.2048 - accuracy:
0.5643 - val_loss: 1.5253 - val_accuracy: 0.4664
Epoch 6/20
75/75 [==============================] - 1s 7ms/step - loss: 1.0644 - accuracy:
0.6169 - val_loss: 1.6115 - val_accuracy: 0.5004
Epoch 7/20
75/75 [==============================] - 1s 7ms/step - loss: 0.9528 - accuracy:
0.6596 - val_loss: 1.2583 - val_accuracy: 0.5844
Epoch 8/20
75/75 [==============================] - 1s 8ms/step - loss: 0.8280 - accuracy:
0.7112 - val_loss: 1.7482 - val_accuracy: 0.5384
Epoch 9/20
75/75 [==============================] - 1s 7ms/step - loss: 0.6933 - accuracy:
0.7561 - val_loss: 1.4774 - val_accuracy: 0.5688
Epoch 10/20
75/75 [==============================] - 1s 7ms/step - loss: 0.5908 - accuracy:
0.7887 - val_loss: 2.5039 - val_accuracy: 0.4500
Epoch 11/20
75/75 [==============================] - 1s 9ms/step - loss: 0.4795 - accuracy:
0.8303 - val_loss: 1.7140 - val_accuracy: 0.5644
Epoch 12/20
75/75 [==============================] - 1s 8ms/step - loss: 0.3476 - accuracy:
0.8779 - val_loss: 1.6995 - val_accuracy: 0.6004
Epoch 13/20
75/75 [==============================] - 1s 7ms/step - loss: 0.2923 - accuracy:
0.8952 - val_loss: 1.8337 - val_accuracy: 0.5920
Epoch 14/20
75/75 [==============================] - 1s 9ms/step - loss: 0.2194 - accuracy:
0.9229 - val_loss: 3.0220 - val_accuracy: 0.4924
Epoch 15/20
75/75 [==============================] - 1s 9ms/step - loss: 0.1549 - accuracy:
0.9487 - val_loss: 2.2830 - val_accuracy: 0.5928
Epoch 16/20
75/75 [==============================] - 1s 10ms/step - loss: 0.1245 - accuracy:
0.9571 - val_loss: 3.1595 - val_accuracy: 0.5208
Epoch 17/20
75/75 [==============================] - 1s 11ms/step - loss: 0.0843 - accuracy:
0.9731 - val_loss: 2.5117 - val_accuracy: 0.5824
Epoch 18/20
75/75 [==============================] - 1s 10ms/step - loss: 0.1025 - accuracy:
0.9659 - val_loss: 3.0090 - val_accuracy: 0.5504
Epoch 19/20
75/75 [==============================] - 1s 8ms/step - loss: 0.1034 - accuracy:
0.9633 - val_loss: 3.2371 - val_accuracy: 0.5496
Epoch 20/20
75/75 [==============================] - 1s 7ms/step - loss: 0.1164 - accuracy:
0.9588 - val_loss: 2.5970 - val_accuracy: 0.5872
```

```
[ ]: # Evaluate the trained model on test set, not used in training or validation
score = model5.evaluate(Xtest, Ytest_enc, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
20/20 [==============================] - 0s 3ms/step - loss: 2.6117 - accuracy:
0.5775
Test loss: 2.6117
Test accuracy: 0.5775
```

```
[ ]: # Plot the history from the training run
plot_results(history5)
```





1.21 Part 16: Rotate the test images

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?

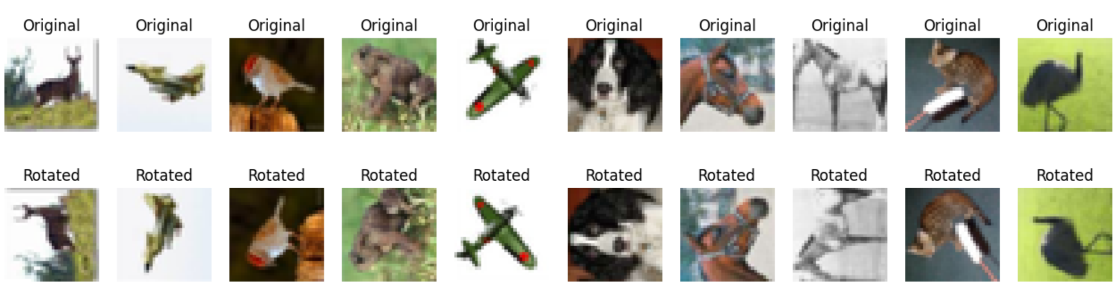
Rotate each test image 90 degrees, the cells are already finished.

Question 21: What is the test accuracy for rotated test images, compared to test images without rotation? Explain the difference in accuracy.

Question 21: The test accuracy has drasticly reduced to 0.2370. The CNN learns based on the specific orientation of the image so if we want the model to perform well for rotated images then we would need to train the model on different orientations of the image.

[ ]:
```
def myrotate(images):

    images_rot = np.rot90(images, axes=(1,2))

    return images_rot
```

[ ]:
```
# 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()
```



[ ]:
```python
# Evaluate the trained model on rotated test set
score = model5.evaluate(Xtest_rotated, Ytest_enc, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
20/20 [==============================] - 0s 4ms/step - loss: 7.5174 - accuracy:
0.2260
Test loss: 7.5174
Test accuracy: 0.2260
```

1.22 Part 17: 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`

See https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator , the .flow(x,y) functionality

Make sure to use different subsets for training and validation when you setup the flows, otherwise you will validate on the same data…

[ ]:
```python
# Get all 60 000 training images again. ImageDataGenerator manages validation␣
↪data on its own
(Xtrain, Ytrain), _ = cifar10.load_data()

# Reduce number of images to 10,000
Xtrain = Xtrain[0:10000]
Ytrain = Ytrain[0:10000]

# Change data type and rescale range
Xtrain = Xtrain.astype('float32')
Xtrain = Xtrain / 127.5 - 1

# Convert labels to hot encoding
Ytrain = to_categorical(Ytrain, 10)
```

[ ]:
```python
# Set up a data generator with on-the-fly data augmentation, 20% validation␣
↪split
# Use a rotation range of 30 degrees, horizontal and vertical flipping
from keras.preprocessing.image import ImageDataGenerator

#Xtrain, Xval, Ytrain, Yval = train_test_split(Xtrain, Ytrain, test_size=0.20,␣
↪random_state=123)
```

```
datagen = ImageDataGenerator(rotation_range=30, horizontal_flip=True,
    vertical_flip=True,validation_split = 0.20)

#datagen = ImageDataGenerator(validation_split = 0.20)

# Setup a flow for training data, assume that we can fit all images into CPU␣
 ↪memory
Xtrain_aug = datagen.flow(Xtrain,Ytrain,batch_size=100,subset = "training")


# Setup a flow for validation data, assume that we can fit all images into CPU␣
 ↪memory
Xval_aug = datagen.flow(Xtrain,Ytrain,batch_size=100,subset = "validation")
```

1.23 Part 18: What about big data?

Question 22: 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?

Question 22: We could load the data in batches rather than loading the entire data at once. This could lead to longer training time.

[ ]:
```
# Plot some augmented images
plot_datagen = datagen.flow(Xtrain, Ytrain, batch_size=1)

plt.figure(figsize=(12,4))
for i in range(18):
    (im, label) = plot_datagen.next()
    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()
```

1.24 Part 19: Train the CNN with images from the generator

See https://keras.io/api/models/model_training_apis/#fit-method for how to use model.fit 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)*(1 - validation_split)/batch_size`

`validation_steps should be set to: len(Xtrain)*validation_split/batch_size`

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

Question 23: 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?

Question 24: What other types of image augmentation can be applied, compared to what we use here?

Question 23: The training accuracy increases slower than compared to without augmentation. It takes more epoch to reach around the same level of accuracy that we see in the "without augementation" model. This could be beacuse the augmentated model has images of different configuration and changes to the image i.e rotation, horizontal and vertical flip. But this makes the model more generalized and robust due to the various changes applied to the images making the model more dynamic to images.

Increase the "epoch" would be needed to perfom more training.

Question 24: Some other augmentation that can be done: zooming into the image, changing the brightness, changing the color, adding a shear etc.

[ ]:
```
# Setup some training parameters
batch_size = 100
epochs = 200
input_shape = (32,32,3)
```

```
# Build model (your best config)
model6 = build_CNN(input_shape, n_conv_layers = 4, n_filters=32,␣
↪n_dense_layers=1, n_nodes=50, learning_rate=0.01 )

validation_split=0.2

steps_per_epoch = len(Xtrain)*(1 - validation_split)/batch_size
validation_steps = len(Xtrain)*validation_split/batch_size


# Train the model using on the fly augmentation
history6 = model6.fit(Xtrain_aug, validation_data= Xval_aug, epochs=epochs,␣
↪batch_size=batch_size, validation_split=validation_split,
                      steps_per_epoch = steps_per_epoch , validation_steps =␣
↪validation_steps)
```

```
Epoch 1/200
80/80 [==============================] - 8s 76ms/step - loss: 2.1090 - accuracy:
0.2170 - val_loss: 2.2775 - val_accuracy: 0.2600
Epoch 2/200
80/80 [==============================] - 6s 73ms/step - loss: 1.8154 - accuracy:
0.3158 - val_loss: 2.6150 - val_accuracy: 0.2095
Epoch 3/200
80/80 [==============================] - 5s 61ms/step - loss: 1.7291 - accuracy:
0.3591 - val_loss: 1.9256 - val_accuracy: 0.3055
Epoch 4/200
80/80 [==============================] - 6s 74ms/step - loss: 1.6544 - accuracy:
0.3873 - val_loss: 2.4487 - val_accuracy: 0.2985
Epoch 5/200
80/80 [==============================] - 5s 60ms/step - loss: 1.5872 - accuracy:
0.4162 - val_loss: 1.9281 - val_accuracy: 0.3440
Epoch 6/200
80/80 [==============================] - 6s 73ms/step - loss: 1.5243 - accuracy:
0.4375 - val_loss: 2.8167 - val_accuracy: 0.2730
Epoch 7/200
80/80 [==============================] - 5s 60ms/step - loss: 1.4779 - accuracy:
0.4638 - val_loss: 1.7788 - val_accuracy: 0.3730
Epoch 8/200
80/80 [==============================] - 6s 81ms/step - loss: 1.4294 - accuracy:
0.4821 - val_loss: 1.7623 - val_accuracy: 0.4015
Epoch 9/200
80/80 [==============================] - 5s 60ms/step - loss: 1.4008 - accuracy:
0.4818 - val_loss: 1.4176 - val_accuracy: 0.4930
Epoch 10/200
80/80 [==============================] - 5s 61ms/step - loss: 1.3546 - accuracy:
0.5073 - val_loss: 1.5479 - val_accuracy: 0.4465
```

```
Epoch 11/200
80/80 [==============================] - 6s 71ms/step - loss: 1.3386 - accuracy:
0.5148 - val_loss: 1.7486 - val_accuracy: 0.4390
Epoch 12/200
80/80 [==============================] - 5s 62ms/step - loss: 1.3026 - accuracy:
0.5235 - val_loss: 1.5237 - val_accuracy: 0.4810
Epoch 13/200
80/80 [==============================] - 6s 77ms/step - loss: 1.2789 - accuracy:
0.5340 - val_loss: 1.7566 - val_accuracy: 0.4200
Epoch 14/200
80/80 [==============================] - 5s 63ms/step - loss: 1.2377 - accuracy:
0.5495 - val_loss: 1.3693 - val_accuracy: 0.4975
Epoch 15/200
80/80 [==============================] - 6s 73ms/step - loss: 1.2219 - accuracy:
0.5562 - val_loss: 1.3468 - val_accuracy: 0.5160
Epoch 16/200
80/80 [==============================] - 5s 62ms/step - loss: 1.1904 - accuracy:
0.5677 - val_loss: 1.3512 - val_accuracy: 0.5275
Epoch 17/200
80/80 [==============================] - 5s 65ms/step - loss: 1.1669 - accuracy:
0.5815 - val_loss: 1.3706 - val_accuracy: 0.5085
Epoch 18/200
80/80 [==============================] - 6s 69ms/step - loss: 1.1599 - accuracy:
0.5822 - val_loss: 1.3643 - val_accuracy: 0.5160
Epoch 19/200
80/80 [==============================] - 6s 74ms/step - loss: 1.1212 - accuracy:
0.5911 - val_loss: 1.6768 - val_accuracy: 0.4770
Epoch 20/200
80/80 [==============================] - 7s 82ms/step - loss: 1.1128 - accuracy:
0.6039 - val_loss: 1.4508 - val_accuracy: 0.5130
Epoch 21/200
80/80 [==============================] - 5s 60ms/step - loss: 1.1187 - accuracy:
0.5964 - val_loss: 1.3068 - val_accuracy: 0.5330
Epoch 22/200
80/80 [==============================] - 5s 64ms/step - loss: 1.0770 - accuracy:
0.6089 - val_loss: 1.2478 - val_accuracy: 0.5785
Epoch 23/200
80/80 [==============================] - 6s 69ms/step - loss: 1.0420 - accuracy:
0.6217 - val_loss: 1.2425 - val_accuracy: 0.5585
Epoch 24/200
80/80 [==============================] - 5s 59ms/step - loss: 1.0432 - accuracy:
0.6298 - val_loss: 1.3385 - val_accuracy: 0.5450
Epoch 25/200
80/80 [==============================] - 6s 74ms/step - loss: 1.0180 - accuracy:
0.6376 - val_loss: 1.3779 - val_accuracy: 0.5380
Epoch 26/200
80/80 [==============================] - 6s 74ms/step - loss: 1.0208 - accuracy:
0.6310 - val_loss: 1.3308 - val_accuracy: 0.5695
```

```
Epoch 27/200
80/80 [==============================] - 5s 61ms/step - loss: 0.9913 - accuracy:
0.6482 - val_loss: 1.6615 - val_accuracy: 0.5680
Epoch 28/200
80/80 [==============================] - 6s 77ms/step - loss: 0.9826 - accuracy:
0.6534 - val_loss: 1.4583 - val_accuracy: 0.5200
Epoch 29/200
80/80 [==============================] - 5s 60ms/step - loss: 0.9515 - accuracy:
0.6631 - val_loss: 1.5173 - val_accuracy: 0.5155
Epoch 30/200
80/80 [==============================] - 8s 95ms/step - loss: 0.9635 - accuracy:
0.6549 - val_loss: 1.2278 - val_accuracy: 0.5850
Epoch 31/200
80/80 [==============================] - 9s 115ms/step - loss: 0.9375 -
accuracy: 0.6704 - val_loss: 1.2179 - val_accuracy: 0.5795
Epoch 32/200
80/80 [==============================] - 5s 62ms/step - loss: 0.9256 - accuracy:
0.6705 - val_loss: 1.1845 - val_accuracy: 0.6230
Epoch 33/200
80/80 [==============================] - 7s 82ms/step - loss: 0.9044 - accuracy:
0.6794 - val_loss: 1.1553 - val_accuracy: 0.6100
Epoch 34/200
80/80 [==============================] - 7s 87ms/step - loss: 0.8983 - accuracy:
0.6773 - val_loss: 1.2193 - val_accuracy: 0.5870
Epoch 35/200
80/80 [==============================] - 8s 93ms/step - loss: 0.8975 - accuracy:
0.6815 - val_loss: 1.1630 - val_accuracy: 0.6125
Epoch 36/200
80/80 [==============================] - 7s 89ms/step - loss: 0.8817 - accuracy:
0.6836 - val_loss: 1.3220 - val_accuracy: 0.5565
Epoch 37/200
80/80 [==============================] - 5s 65ms/step - loss: 0.9022 - accuracy:
0.6796 - val_loss: 1.2216 - val_accuracy: 0.6045
Epoch 38/200
80/80 [==============================] - 5s 61ms/step - loss: 0.8718 - accuracy:
0.6917 - val_loss: 1.0685 - val_accuracy: 0.6240
Epoch 39/200
80/80 [==============================] - 6s 73ms/step - loss: 0.8459 - accuracy:
0.6980 - val_loss: 1.1094 - val_accuracy: 0.6135
Epoch 40/200
80/80 [==============================] - 5s 59ms/step - loss: 0.8382 - accuracy:
0.7017 - val_loss: 1.1266 - val_accuracy: 0.6235
Epoch 41/200
80/80 [==============================] - 6s 81ms/step - loss: 0.8325 - accuracy:
0.6991 - val_loss: 1.2480 - val_accuracy: 0.6030
Epoch 42/200
80/80 [==============================] - 5s 59ms/step - loss: 0.8338 - accuracy:
0.7057 - val_loss: 1.1882 - val_accuracy: 0.6030
```

```
Epoch 43/200
80/80 [==============================] - 6s 74ms/step - loss: 0.8114 - accuracy:
0.7082 - val_loss: 1.2011 - val_accuracy: 0.6095
Epoch 44/200
80/80 [==============================] - 6s 78ms/step - loss: 0.8019 - accuracy:
0.7151 - val_loss: 1.0780 - val_accuracy: 0.6440
Epoch 45/200
80/80 [==============================] - 5s 58ms/step - loss: 0.7998 - accuracy:
0.7125 - val_loss: 1.1179 - val_accuracy: 0.6225
Epoch 46/200
80/80 [==============================] - 6s 70ms/step - loss: 0.7820 - accuracy:
0.7264 - val_loss: 1.0638 - val_accuracy: 0.6495
Epoch 47/200
80/80 [==============================] - 5s 61ms/step - loss: 0.7847 - accuracy:
0.7270 - val_loss: 1.0809 - val_accuracy: 0.6390
Epoch 48/200
80/80 [==============================] - 6s 73ms/step - loss: 0.7771 - accuracy:
0.7278 - val_loss: 1.0900 - val_accuracy: 0.6410
Epoch 49/200
80/80 [==============================] - 5s 66ms/step - loss: 0.7574 - accuracy:
0.7324 - val_loss: 1.2336 - val_accuracy: 0.6110
Epoch 50/200
80/80 [==============================] - 6s 73ms/step - loss: 0.7556 - accuracy:
0.7340 - val_loss: 1.4945 - val_accuracy: 0.5700
Epoch 51/200
80/80 [==============================] - 5s 60ms/step - loss: 0.7617 - accuracy:
0.7284 - val_loss: 1.2042 - val_accuracy: 0.6190
Epoch 52/200
80/80 [==============================] - 6s 76ms/step - loss: 0.7580 - accuracy:
0.7254 - val_loss: 1.0852 - val_accuracy: 0.6325
Epoch 53/200
80/80 [==============================] - 8s 97ms/step - loss: 0.7399 - accuracy:
0.7339 - val_loss: 1.1216 - val_accuracy: 0.6350
Epoch 54/200
80/80 [==============================] - 8s 102ms/step - loss: 0.7344 -
accuracy: 0.7414 - val_loss: 1.1032 - val_accuracy: 0.6405
Epoch 55/200
80/80 [==============================] - 8s 102ms/step - loss: 0.7197 -
accuracy: 0.7501 - val_loss: 1.1889 - val_accuracy: 0.6145
Epoch 56/200
80/80 [==============================] - 9s 107ms/step - loss: 0.7259 -
accuracy: 0.7442 - val_loss: 1.2697 - val_accuracy: 0.6160
Epoch 57/200
80/80 [==============================] - 10s 119ms/step - loss: 0.7151 -
accuracy: 0.7445 - val_loss: 1.0706 - val_accuracy: 0.6435
Epoch 58/200
80/80 [==============================] - 8s 96ms/step - loss: 0.7137 - accuracy:
0.7516 - val_loss: 1.1748 - val_accuracy: 0.6115
```

```
Epoch 59/200
80/80 [==============================] - 5s 60ms/step - loss: 0.6834 - accuracy:
0.7515 - val_loss: 1.1261 - val_accuracy: 0.6405
Epoch 60/200
80/80 [==============================] - 6s 74ms/step - loss: 0.6963 - accuracy:
0.7467 - val_loss: 1.1666 - val_accuracy: 0.6205
Epoch 61/200
80/80 [==============================] - 7s 83ms/step - loss: 0.6836 - accuracy:
0.7561 - val_loss: 1.0904 - val_accuracy: 0.6515
Epoch 62/200
80/80 [==============================] - 8s 100ms/step - loss: 0.6740 -
accuracy: 0.7560 - val_loss: 1.2080 - val_accuracy: 0.6180
Epoch 63/200
80/80 [==============================] - 8s 106ms/step - loss: 0.6788 -
accuracy: 0.7596 - val_loss: 1.2024 - val_accuracy: 0.6210
Epoch 64/200
80/80 [==============================] - 8s 104ms/step - loss: 0.6550 -
accuracy: 0.7675 - val_loss: 1.2511 - val_accuracy: 0.6140
Epoch 65/200
80/80 [==============================] - 6s 79ms/step - loss: 0.6709 - accuracy:
0.7640 - val_loss: 1.2009 - val_accuracy: 0.6390
Epoch 66/200
80/80 [==============================] - 6s 75ms/step - loss: 0.6445 - accuracy:
0.7744 - val_loss: 1.1158 - val_accuracy: 0.6330
Epoch 67/200
80/80 [==============================] - 5s 65ms/step - loss: 0.6515 - accuracy:
0.7688 - val_loss: 1.2548 - val_accuracy: 0.6140
Epoch 68/200
80/80 [==============================] - 5s 68ms/step - loss: 0.6336 - accuracy:
0.7790 - val_loss: 1.1540 - val_accuracy: 0.6300
Epoch 69/200
80/80 [==============================] - 6s 74ms/step - loss: 0.6483 - accuracy:
0.7742 - val_loss: 1.1424 - val_accuracy: 0.6395
Epoch 70/200
80/80 [==============================] - 7s 83ms/step - loss: 0.6338 - accuracy:
0.7735 - val_loss: 1.1621 - val_accuracy: 0.6485
Epoch 71/200
80/80 [==============================] - 9s 118ms/step - loss: 0.6356 -
accuracy: 0.7724 - val_loss: 1.1935 - val_accuracy: 0.6330
Epoch 72/200
80/80 [==============================] - 7s 86ms/step - loss: 0.6150 - accuracy:
0.7825 - val_loss: 1.1455 - val_accuracy: 0.6425
Epoch 73/200
80/80 [==============================] - 8s 103ms/step - loss: 0.6123 -
accuracy: 0.7804 - val_loss: 1.2026 - val_accuracy: 0.6255
Epoch 74/200
80/80 [==============================] - 7s 91ms/step - loss: 0.6102 - accuracy:
0.7820 - val_loss: 1.1384 - val_accuracy: 0.6520
```

```
Epoch 75/200
80/80 [==============================] - 8s 95ms/step - loss: 0.5934 - accuracy:
0.7861 - val_loss: 1.2897 - val_accuracy: 0.6225
Epoch 76/200
80/80 [==============================] - 7s 91ms/step - loss: 0.6074 - accuracy:
0.7840 - val_loss: 1.1908 - val_accuracy: 0.6425
Epoch 77/200
80/80 [==============================] - 6s 74ms/step - loss: 0.6005 - accuracy:
0.7853 - val_loss: 1.1855 - val_accuracy: 0.6350
Epoch 78/200
80/80 [==============================] - 5s 60ms/step - loss: 0.5920 - accuracy:
0.7910 - val_loss: 1.2510 - val_accuracy: 0.6260
Epoch 79/200
80/80 [==============================] - 6s 73ms/step - loss: 0.5766 - accuracy:
0.7931 - val_loss: 1.1505 - val_accuracy: 0.6460
Epoch 80/200
80/80 [==============================] - 5s 63ms/step - loss: 0.5982 - accuracy:
0.7881 - val_loss: 1.1412 - val_accuracy: 0.6615
Epoch 81/200
80/80 [==============================] - 6s 76ms/step - loss: 0.5797 - accuracy:
0.7878 - val_loss: 1.3494 - val_accuracy: 0.5975
Epoch 82/200
80/80 [==============================] - 6s 76ms/step - loss: 0.5664 - accuracy:
0.8054 - val_loss: 1.3474 - val_accuracy: 0.6180
Epoch 83/200
80/80 [==============================] - 11s 144ms/step - loss: 0.5567 -
accuracy: 0.8070 - val_loss: 1.1548 - val_accuracy: 0.6515
Epoch 84/200
80/80 [==============================] - 7s 87ms/step - loss: 0.5560 - accuracy:
0.8035 - val_loss: 1.2529 - val_accuracy: 0.6355
Epoch 85/200
80/80 [==============================] - 8s 99ms/step - loss: 0.5425 - accuracy:
0.8109 - val_loss: 1.2703 - val_accuracy: 0.6270
Epoch 86/200
80/80 [==============================] - 8s 106ms/step - loss: 0.5467 -
accuracy: 0.8074 - val_loss: 1.3261 - val_accuracy: 0.6300
Epoch 87/200
80/80 [==============================] - 8s 105ms/step - loss: 0.5499 -
accuracy: 0.8080 - val_loss: 1.1101 - val_accuracy: 0.6695
Epoch 88/200
80/80 [==============================] - 8s 95ms/step - loss: 0.5429 - accuracy:
0.8101 - val_loss: 1.2124 - val_accuracy: 0.6405
Epoch 89/200
80/80 [==============================] - 7s 87ms/step - loss: 0.5546 - accuracy:
0.8043 - val_loss: 1.1692 - val_accuracy: 0.6490
Epoch 90/200
80/80 [==============================] - 5s 63ms/step - loss: 0.5330 - accuracy:
0.8073 - val_loss: 1.3004 - val_accuracy: 0.6315
```

```
Epoch 91/200
80/80 [==============================] - 8s 97ms/step - loss: 0.5491 - accuracy:
0.8012 - val_loss: 1.1525 - val_accuracy: 0.6565
Epoch 92/200
80/80 [==============================] - 5s 59ms/step - loss: 0.5315 - accuracy:
0.8152 - val_loss: 1.1386 - val_accuracy: 0.6690
Epoch 93/200
80/80 [==============================] - 7s 83ms/step - loss: 0.5266 - accuracy:
0.8100 - val_loss: 1.2107 - val_accuracy: 0.6620
Epoch 94/200
80/80 [==============================] - 5s 61ms/step - loss: 0.5262 - accuracy:
0.8165 - val_loss: 1.1562 - val_accuracy: 0.6440
Epoch 95/200
80/80 [==============================] - 9s 118ms/step - loss: 0.5015 -
accuracy: 0.8195 - val_loss: 1.2967 - val_accuracy: 0.6250
Epoch 96/200
80/80 [==============================] - 5s 63ms/step - loss: 0.5234 - accuracy:
0.8158 - val_loss: 1.1493 - val_accuracy: 0.6560
Epoch 97/200
80/80 [==============================] - 6s 73ms/step - loss: 0.4977 - accuracy:
0.8244 - val_loss: 1.2194 - val_accuracy: 0.6460
Epoch 98/200
80/80 [==============================] - 6s 73ms/step - loss: 0.5147 - accuracy:
0.8210 - val_loss: 1.3283 - val_accuracy: 0.6290
Epoch 99/200
80/80 [==============================] - 10s 128ms/step - loss: 0.5075 -
accuracy: 0.8163 - val_loss: 1.1617 - val_accuracy: 0.6460
Epoch 100/200
80/80 [==============================] - 5s 61ms/step - loss: 0.4861 - accuracy:
0.8276 - val_loss: 1.2227 - val_accuracy: 0.6385
Epoch 101/200
80/80 [==============================] - 6s 74ms/step - loss: 0.4945 - accuracy:
0.8240 - val_loss: 1.3707 - val_accuracy: 0.6220
Epoch 102/200
80/80 [==============================] - 5s 62ms/step - loss: 0.5046 - accuracy:
0.8185 - val_loss: 1.2233 - val_accuracy: 0.6530
Epoch 103/200
80/80 [==============================] - 7s 84ms/step - loss: 0.4933 - accuracy:
0.8239 - val_loss: 1.2325 - val_accuracy: 0.6565
Epoch 104/200
80/80 [==============================] - 5s 61ms/step - loss: 0.4835 - accuracy:
0.8261 - val_loss: 1.1974 - val_accuracy: 0.6530
Epoch 105/200
80/80 [==============================] - 6s 74ms/step - loss: 0.4927 - accuracy:
0.8229 - val_loss: 1.1727 - val_accuracy: 0.6755
Epoch 106/200
80/80 [==============================] - 6s 73ms/step - loss: 0.4783 - accuracy:
0.8273 - val_loss: 1.2896 - val_accuracy: 0.6270
```

```
Epoch 107/200
80/80 [==============================] - 5s 61ms/step - loss: 0.4792 - accuracy:
0.8316 - val_loss: 1.2733 - val_accuracy: 0.6490
Epoch 108/200
80/80 [==============================] - 5s 65ms/step - loss: 0.4705 - accuracy:
0.8298 - val_loss: 1.2157 - val_accuracy: 0.6555
Epoch 109/200
80/80 [==============================] - 6s 71ms/step - loss: 0.4615 - accuracy:
0.8386 - val_loss: 1.2476 - val_accuracy: 0.6505
Epoch 110/200
80/80 [==============================] - 5s 60ms/step - loss: 0.4752 - accuracy:
0.8313 - val_loss: 1.2432 - val_accuracy: 0.6445
Epoch 111/200
80/80 [==============================] - 6s 75ms/step - loss: 0.4541 - accuracy:
0.8399 - val_loss: 1.1725 - val_accuracy: 0.6630
Epoch 112/200
80/80 [==============================] - 5s 59ms/step - loss: 0.4627 - accuracy:
0.8353 - val_loss: 1.3356 - val_accuracy: 0.6275
Epoch 113/200
80/80 [==============================] - 6s 73ms/step - loss: 0.4407 - accuracy:
0.8464 - val_loss: 1.3550 - val_accuracy: 0.6425
Epoch 114/200
80/80 [==============================] - 5s 60ms/step - loss: 0.4690 - accuracy:
0.8339 - val_loss: 1.2764 - val_accuracy: 0.6430
Epoch 115/200
80/80 [==============================] - 5s 66ms/step - loss: 0.4476 - accuracy:
0.8430 - val_loss: 1.4014 - val_accuracy: 0.6040
Epoch 116/200
80/80 [==============================] - 5s 60ms/step - loss: 0.4387 - accuracy:
0.8413 - val_loss: 1.2891 - val_accuracy: 0.6585
Epoch 117/200
80/80 [==============================] - 6s 75ms/step - loss: 0.4499 - accuracy:
0.8432 - val_loss: 1.2957 - val_accuracy: 0.6450
Epoch 118/200
80/80 [==============================] - 5s 60ms/step - loss: 0.4460 - accuracy:
0.8413 - val_loss: 1.3584 - val_accuracy: 0.6365
Epoch 119/200
80/80 [==============================] - 6s 76ms/step - loss: 0.4395 - accuracy:
0.8446 - val_loss: 1.2295 - val_accuracy: 0.6555
Epoch 120/200
80/80 [==============================] - 5s 61ms/step - loss: 0.4415 - accuracy:
0.8429 - val_loss: 1.2514 - val_accuracy: 0.6600
Epoch 121/200
80/80 [==============================] - 6s 74ms/step - loss: 0.4154 - accuracy:
0.8550 - val_loss: 1.3618 - val_accuracy: 0.6395
Epoch 122/200
80/80 [==============================] - 5s 63ms/step - loss: 0.4278 - accuracy:
0.8511 - val_loss: 1.2583 - val_accuracy: 0.6465
```

```
Epoch 123/200
80/80 [==============================] - 6s 74ms/step - loss: 0.4374 - accuracy:
0.8462 - val_loss: 1.3482 - val_accuracy: 0.6495
Epoch 124/200
80/80 [==============================] - 5s 62ms/step - loss: 0.4310 - accuracy:
0.8489 - val_loss: 1.1962 - val_accuracy: 0.6680
Epoch 125/200
80/80 [==============================] - 6s 76ms/step - loss: 0.3957 - accuracy:
0.8581 - val_loss: 1.2539 - val_accuracy: 0.6655
Epoch 126/200
80/80 [==============================] - 5s 60ms/step - loss: 0.4227 - accuracy:
0.8519 - val_loss: 1.3497 - val_accuracy: 0.6425
Epoch 127/200
80/80 [==============================] - 6s 72ms/step - loss: 0.4130 - accuracy:
0.8526 - val_loss: 1.2820 - val_accuracy: 0.6470
Epoch 128/200
80/80 [==============================] - 5s 62ms/step - loss: 0.4100 - accuracy:
0.8564 - val_loss: 1.2911 - val_accuracy: 0.6645
Epoch 129/200
80/80 [==============================] - 6s 75ms/step - loss: 0.4044 - accuracy:
0.8616 - val_loss: 1.2827 - val_accuracy: 0.6580
Epoch 130/200
80/80 [==============================] - 6s 74ms/step - loss: 0.4006 - accuracy:
0.8536 - val_loss: 1.5713 - val_accuracy: 0.6125
Epoch 131/200
80/80 [==============================] - 5s 60ms/step - loss: 0.4037 - accuracy:
0.8553 - val_loss: 1.2534 - val_accuracy: 0.6495
Epoch 132/200
80/80 [==============================] - 6s 74ms/step - loss: 0.4034 - accuracy:
0.8590 - val_loss: 1.5546 - val_accuracy: 0.6260
Epoch 133/200
80/80 [==============================] - 5s 61ms/step - loss: 0.4048 - accuracy:
0.8562 - val_loss: 1.3575 - val_accuracy: 0.6405
Epoch 134/200
80/80 [==============================] - 6s 75ms/step - loss: 0.3781 - accuracy:
0.8666 - val_loss: 1.3252 - val_accuracy: 0.6360
Epoch 135/200
80/80 [==============================] - 8s 95ms/step - loss: 0.3938 - accuracy:
0.8576 - val_loss: 1.2603 - val_accuracy: 0.6570
Epoch 136/200
80/80 [==============================] - 7s 82ms/step - loss: 0.3911 - accuracy:
0.8586 - val_loss: 1.2699 - val_accuracy: 0.6595
Epoch 137/200
80/80 [==============================] - 7s 87ms/step - loss: 0.4014 - accuracy:
0.8575 - val_loss: 1.3615 - val_accuracy: 0.6420
Epoch 138/200
80/80 [==============================] - 7s 85ms/step - loss: 0.3758 - accuracy:
0.8708 - val_loss: 1.3087 - val_accuracy: 0.6500
```

```
Epoch 139/200
80/80 [==============================] - 6s 80ms/step - loss: 0.3684 - accuracy:
0.8724 - val_loss: 1.4060 - val_accuracy: 0.6410
Epoch 140/200
80/80 [==============================] - 7s 87ms/step - loss: 0.3796 - accuracy:
0.8675 - val_loss: 1.3656 - val_accuracy: 0.6470
Epoch 141/200
80/80 [==============================] - 10s 123ms/step - loss: 0.3671 -
accuracy: 0.8719 - val_loss: 1.2605 - val_accuracy: 0.6600
Epoch 142/200
80/80 [==============================] - 7s 89ms/step - loss: 0.3734 - accuracy:
0.8670 - val_loss: 1.3535 - val_accuracy: 0.6420
Epoch 143/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3806 - accuracy:
0.8640 - val_loss: 1.3989 - val_accuracy: 0.6430
Epoch 144/200
80/80 [==============================] - 10s 119ms/step - loss: 0.3818 -
accuracy: 0.8646 - val_loss: 1.3071 - val_accuracy: 0.6510
Epoch 145/200
80/80 [==============================] - 5s 62ms/step - loss: 0.3914 - accuracy:
0.8609 - val_loss: 1.2956 - val_accuracy: 0.6610
Epoch 146/200
80/80 [==============================] - 7s 85ms/step - loss: 0.3684 - accuracy:
0.8710 - val_loss: 1.3497 - val_accuracy: 0.6520
Epoch 147/200
80/80 [==============================] - 7s 93ms/step - loss: 0.3747 - accuracy:
0.8715 - val_loss: 1.2621 - val_accuracy: 0.6765
Epoch 148/200
80/80 [==============================] - 7s 90ms/step - loss: 0.3714 - accuracy:
0.8684 - val_loss: 1.4103 - val_accuracy: 0.6465
Epoch 149/200
80/80 [==============================] - 7s 82ms/step - loss: 0.3480 - accuracy:
0.8761 - val_loss: 1.4506 - val_accuracy: 0.6470
Epoch 150/200
80/80 [==============================] - 8s 93ms/step - loss: 0.3539 - accuracy:
0.8786 - val_loss: 1.2849 - val_accuracy: 0.6665
Epoch 151/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3674 - accuracy:
0.8750 - val_loss: 1.4674 - val_accuracy: 0.6450
Epoch 152/200
80/80 [==============================] - 5s 60ms/step - loss: 0.3515 - accuracy:
0.8755 - val_loss: 1.4017 - val_accuracy: 0.6520
Epoch 153/200
80/80 [==============================] - 7s 88ms/step - loss: 0.3643 - accuracy:
0.8684 - val_loss: 1.3985 - val_accuracy: 0.6415
Epoch 154/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3594 - accuracy:
0.8711 - val_loss: 1.2852 - val_accuracy: 0.6535
```

```
Epoch 155/200
80/80 [==============================] - 5s 63ms/step - loss: 0.3483 - accuracy:
0.8765 - val_loss: 1.3791 - val_accuracy: 0.6530
Epoch 156/200
80/80 [==============================] - 5s 68ms/step - loss: 0.3372 - accuracy:
0.8775 - val_loss: 1.2681 - val_accuracy: 0.6745
Epoch 157/200
80/80 [==============================] - 5s 65ms/step - loss: 0.3355 - accuracy:
0.8840 - val_loss: 1.4600 - val_accuracy: 0.6365
Epoch 158/200
80/80 [==============================] - 5s 61ms/step - loss: 0.3523 - accuracy:
0.8740 - val_loss: 1.3980 - val_accuracy: 0.6580
Epoch 159/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3467 - accuracy:
0.8776 - val_loss: 1.4427 - val_accuracy: 0.6480
Epoch 160/200
80/80 [==============================] - 5s 66ms/step - loss: 0.3398 - accuracy:
0.8816 - val_loss: 1.3207 - val_accuracy: 0.6720
Epoch 161/200
80/80 [==============================] - 6s 69ms/step - loss: 0.3284 - accuracy:
0.8851 - val_loss: 1.4890 - val_accuracy: 0.6385
Epoch 162/200
80/80 [==============================] - 5s 62ms/step - loss: 0.3254 - accuracy:
0.8851 - val_loss: 1.4538 - val_accuracy: 0.6525
Epoch 163/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3203 - accuracy:
0.8898 - val_loss: 1.3201 - val_accuracy: 0.6780
Epoch 164/200
80/80 [==============================] - 5s 62ms/step - loss: 0.3320 - accuracy:
0.8834 - val_loss: 1.3457 - val_accuracy: 0.6635
Epoch 165/200
80/80 [==============================] - 6s 75ms/step - loss: 0.3430 - accuracy:
0.8799 - val_loss: 1.3907 - val_accuracy: 0.6485
Epoch 166/200
80/80 [==============================] - 5s 61ms/step - loss: 0.3404 - accuracy:
0.8785 - val_loss: 1.3546 - val_accuracy: 0.6625
Epoch 167/200
80/80 [==============================] - 6s 76ms/step - loss: 0.3221 - accuracy:
0.8838 - val_loss: 1.4613 - val_accuracy: 0.6550
Epoch 168/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3257 - accuracy:
0.8864 - val_loss: 1.4235 - val_accuracy: 0.6610
Epoch 169/200
80/80 [==============================] - 5s 63ms/step - loss: 0.3177 - accuracy:
0.8866 - val_loss: 1.4215 - val_accuracy: 0.6505
Epoch 170/200
80/80 [==============================] - 7s 83ms/step - loss: 0.3115 - accuracy:
0.8894 - val_loss: 1.3847 - val_accuracy: 0.6650
```

```
Epoch 171/200
80/80 [==============================] - 5s 62ms/step - loss: 0.3360 - accuracy:
0.8825 - val_loss: 1.4369 - val_accuracy: 0.6350
Epoch 172/200
80/80 [==============================] - 5s 66ms/step - loss: 0.3367 - accuracy:
0.8815 - val_loss: 1.4521 - val_accuracy: 0.6435
Epoch 173/200
80/80 [==============================] - 6s 71ms/step - loss: 0.3279 - accuracy:
0.8831 - val_loss: 1.3906 - val_accuracy: 0.6570
Epoch 174/200
80/80 [==============================] - 5s 61ms/step - loss: 0.3247 - accuracy:
0.8874 - val_loss: 1.3671 - val_accuracy: 0.6745
Epoch 175/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3277 - accuracy:
0.8867 - val_loss: 1.4539 - val_accuracy: 0.6450
Epoch 176/200
80/80 [==============================] - 5s 61ms/step - loss: 0.3071 - accuracy:
0.8895 - val_loss: 1.5081 - val_accuracy: 0.6380
Epoch 177/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3044 - accuracy:
0.8914 - val_loss: 1.3445 - val_accuracy: 0.6605
Epoch 178/200
80/80 [==============================] - 8s 97ms/step - loss: 0.2955 - accuracy:
0.8971 - val_loss: 1.5642 - val_accuracy: 0.6515
Epoch 179/200
80/80 [==============================] - 8s 100ms/step - loss: 0.3289 -
accuracy: 0.8832 - val_loss: 1.3703 - val_accuracy: 0.6570
Epoch 180/200
80/80 [==============================] - 7s 90ms/step - loss: 0.2961 - accuracy:
0.8955 - val_loss: 1.4424 - val_accuracy: 0.6565
Epoch 181/200
80/80 [==============================] - 5s 69ms/step - loss: 0.2918 - accuracy:
0.8969 - val_loss: 1.4329 - val_accuracy: 0.6545
Epoch 182/200
80/80 [==============================] - 5s 60ms/step - loss: 0.3135 - accuracy:
0.8876 - val_loss: 1.5895 - val_accuracy: 0.6395
Epoch 183/200
80/80 [==============================] - 6s 74ms/step - loss: 0.2918 - accuracy:
0.8981 - val_loss: 1.4807 - val_accuracy: 0.6625
Epoch 184/200
80/80 [==============================] - 6s 75ms/step - loss: 0.3159 - accuracy:
0.8911 - val_loss: 1.5512 - val_accuracy: 0.6355
Epoch 185/200
80/80 [==============================] - 5s 61ms/step - loss: 0.2867 - accuracy:
0.9036 - val_loss: 1.4667 - val_accuracy: 0.6445
Epoch 186/200
80/80 [==============================] - 5s 63ms/step - loss: 0.2952 - accuracy:
0.8939 - val_loss: 1.4428 - val_accuracy: 0.6675
```

```
Epoch 187/200
80/80 [==============================] - 7s 85ms/step - loss: 0.2874 - accuracy:
0.8978 - val_loss: 1.4835 - val_accuracy: 0.6480
Epoch 188/200
80/80 [==============================] - 5s 63ms/step - loss: 0.3186 - accuracy:
0.8892 - val_loss: 1.4058 - val_accuracy: 0.6625
Epoch 189/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3005 - accuracy:
0.8931 - val_loss: 1.4398 - val_accuracy: 0.6545
Epoch 190/200
80/80 [==============================] - 6s 75ms/step - loss: 0.2932 - accuracy:
0.8906 - val_loss: 1.5475 - val_accuracy: 0.6455
Epoch 191/200
80/80 [==============================] - 5s 65ms/step - loss: 0.2930 - accuracy:
0.8966 - val_loss: 1.5266 - val_accuracy: 0.6555
Epoch 192/200
80/80 [==============================] - 6s 70ms/step - loss: 0.2878 - accuracy:
0.8981 - val_loss: 1.4595 - val_accuracy: 0.6565
Epoch 193/200
80/80 [==============================] - 5s 63ms/step - loss: 0.2862 - accuracy:
0.9013 - val_loss: 1.3470 - val_accuracy: 0.6660
Epoch 194/200
80/80 [==============================] - 6s 75ms/step - loss: 0.2833 - accuracy:
0.9001 - val_loss: 1.4817 - val_accuracy: 0.6510
Epoch 195/200
80/80 [==============================] - 7s 87ms/step - loss: 0.2840 - accuracy:
0.8984 - val_loss: 1.5376 - val_accuracy: 0.6520
Epoch 196/200
80/80 [==============================] - 5s 64ms/step - loss: 0.2793 - accuracy:
0.9007 - val_loss: 1.4867 - val_accuracy: 0.6605
Epoch 197/200
80/80 [==============================] - 7s 88ms/step - loss: 0.2803 - accuracy:
0.9003 - val_loss: 1.4816 - val_accuracy: 0.6520
Epoch 198/200
80/80 [==============================] - 5s 62ms/step - loss: 0.2878 - accuracy:
0.8976 - val_loss: 1.4784 - val_accuracy: 0.6595
Epoch 199/200
80/80 [==============================] - 7s 83ms/step - loss: 0.2896 - accuracy:
0.8995 - val_loss: 1.4029 - val_accuracy: 0.6615
Epoch 200/200
80/80 [==============================] - 5s 62ms/step - loss: 0.2732 - accuracy:
0.9038 - val_loss: 1.4537 - val_accuracy: 0.6770
```

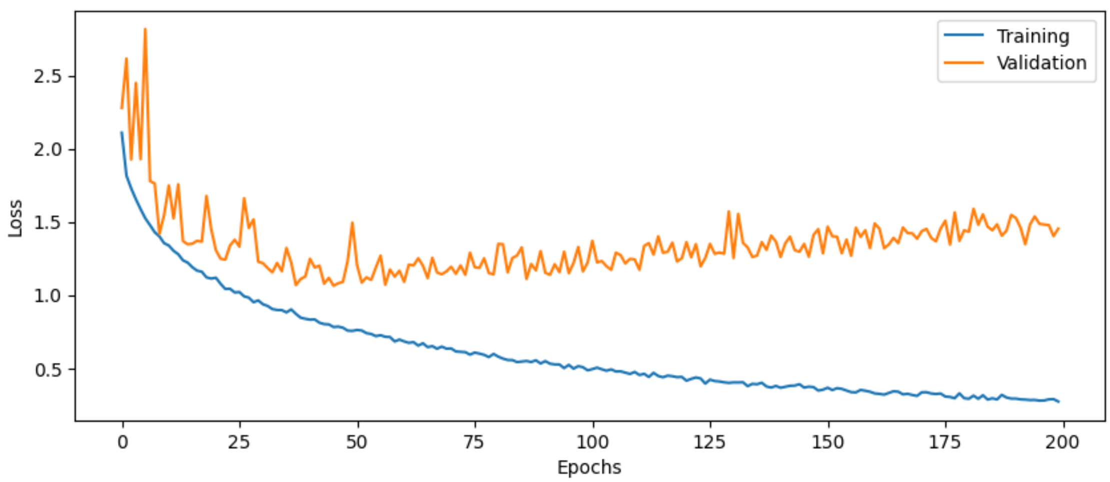
[ ]:
```
# Check if there is still a big difference in accuracy for original and rotated
↪test images

# Evaluate the trained model on original test set
```

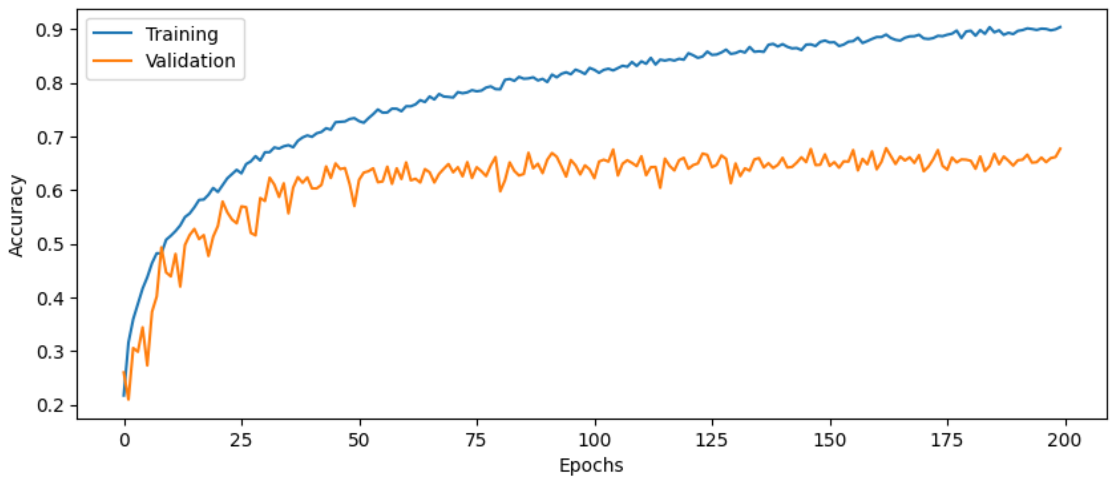
```python
score = model6.evaluate(Xtest, Ytest_enc, batch_size = batch_size, verbose=0)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

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

```
Test loss: 1.4951
Test accuracy: 0.6685
Test loss: 4.2030
Test accuracy: 0.3345
```

[ ]:

```python
# Plot the history from the training run
plot_results(history6)
```





1.25 Part 20: Plot misclassified images

Lets plot some images where the CNN performed badly, these cells are already finished.

[ ]:
```
# Find misclassified images
y_pred=model6.predict(Xtest)
y_pred=np.argmax(y_pred,axis=1)

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

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

```
63/63 [==============================] - 1s 6ms/step
```

[ ]:
```
# 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[label_pred]))
plt.show()
```

1.26 Part 21: Testing on another size

Question 25: 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?

Question 26: 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?

Question 25: No, this is beacuse the model is end with a fully connected network, so it requires a fixed input size due to the flattening operation that maps it to a 1D vector.

Question 26: Yes, this can be doing using only a fully convolutional network i.e without any dense layers

1.27 Part 22: 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()`

Question 27: How many convolutional layers does ResNet50 have?

Question 28: How many trainable parameters does the ResNet50 network have?

Question 29: What is the size of the images that ResNet50 expects as input?

Question 30: Using the answer to question 28, explain why the second derivative is seldom used when training deep networks.

Apply the pre-trained CNN 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 laboration.

See https://keras.io/api/applications/ and https://keras.io/api/applications/resnet/#resnet50-function

Useful functions

`image.load_img` in tensorflow.keras.preprocessing `image.img_to_array` in tensorflow.keras.preprocessing

`ResNet50` in tensorflow.keras.applications.resnet50

`preprocess_input` in tensorflow.keras.applications.resnet50

`decode_predictions` in tensorflow.keras.applications.resnet50

`expand_dims` in numpy

Question 27: 50 convolutional layers

Question 28: 25583592 trainable parameters

Question 29: (224, 224, 3)

Question 30: We dont use second derivative to learn cause to do it we need the hessian matrix which for 25583592 trainable parameter for examples will be 25583592 x 25583592 matrix and that is too big to store in memory.

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

```
Mounted at /content/drive
```

[ ]:
```
# Your code for using pre-trained ResNet 50 on 5 color images of your choice.
# The preprocessing should transform the image to a size that is expected by␣
 ↪the CNN.
from keras.applications import ResNet50
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.applications.resnet50 import preprocess_input ,␣
 ↪decode_predictions


model7 = ResNet50()

#model7.summary()

%cd "/content/drive/MyDrive/"
```

```
/content/drive/MyDrive
```

Image-1: Iron Man

Image-2: Apple

Image-3: Surfing

Image-4: Sun

Image-5: Rocket

[ ]:
```
image1 = load_img("iron-man.webp", target_size = (224,224))
image1=image1.resize((224,224))
```

```
input_arr_1 = img_to_array(image1)
input_arr_1 = np.expand_dims(input_arr_1,axis = 0)

image2 = load_img("apple.webp", target_size = (224,224))
image2=image2.resize((224,224))
input_arr_2 = img_to_array(image2)
input_arr_2 = np.expand_dims(input_arr_2,axis = 0)


image3 = load_img("surf.webp", target_size = (224,224))
image3=image3.resize((224,224))
input_arr_3 = img_to_array(image3)
input_arr_3 = np.expand_dims(input_arr_3,axis = 0)


image4 = load_img("sun.jpeg", target_size = (224,224))
image4=image4.resize((224,224))
input_arr_4 = img_to_array(image4)
input_arr_4 = np.expand_dims(input_arr_4,axis = 0)


image5 = load_img("rocket.webp", target_size = (224,224))
image5=image5.resize((224,224))
input_arr_5 = img_to_array(image5)
input_arr_5 = np.expand_dims(input_arr_5,axis = 0)
```

[ ]:
```
score1 = model7.predict(input_arr_1)
decode_predictions(score1) #Image-1
```

```
1/1 [==============================] - 1s 847ms/step
```

[ ]:
```
[[('n03146219', 'cuirass', 0.80024636),
  ('n02895154', 'breastplate', 0.18907034),
  ('n03000247', 'chain_mail', 0.003918813),
  ('n04192698', 'shield', 0.0021846096),
  ('n03379051', 'football_helmet', 0.0019023493)]]
```

[ ]:
```
score2 = model7.predict(input_arr_2)
decode_predictions(score2) #Image-2
```

```
1/1 [==============================] - 0s 22ms/step
```

[ ]:
```
[[('n07749582', 'lemon', 0.27165836),
  ('n07742313', 'Granny_Smith', 0.26778814),
  ('n07747607', 'orange', 0.10203416),
  ('n07745940', 'strawberry', 0.051424935),
  ('n04522168', 'vase', 0.04739114)]]
```

[ ]:
```
score3 = model7.predict(input_arr_3)
decode_predictions(score3) #Image-3
```

```
1/1 [==============================] - 0s 21ms/step
```

[ ]:
```
[[('n04456115', 'torch', 0.36744487),
  ('n03729826', 'matchstick', 0.21474075),
  ('n03929660', 'pick', 0.07949964),
  ('n03388043', 'fountain', 0.06841073),
  ('n03666591', 'lighter', 0.044225983)]]
```

[ ]:
```
score4 = model7.predict(input_arr_4)
decode_predictions(score4) #Image-4
```

```
1/1 [==============================] - 0s 22ms/step
```

[ ]:
```
[[('n06874185', 'traffic_light', 0.503199),
  ('n04009552', 'projector', 0.3440176),
  ('n04286575', 'spotlight', 0.1258798),
  ('n01930112', 'nematode', 0.008039685),
  ('n03782006', 'monitor', 0.0029325872)]]
```

[ ]:
```
score5 = model7.predict(input_arr_5)
decode_predictions(score5) #Image-5
```

```
1/1 [==============================] - 0s 22ms/step
```

[ ]:
```
[[('n02783161', 'ballpoint', 0.6085586),
  ('n03773504', 'missile', 0.07037285),
  ('n04008634', 'projectile', 0.046716996),
  ('n04116512', 'rubber_eraser', 0.04079768),
  ('n04367480', 'swab', 0.019980296)]]
```

The model doesnt give an exactly right prediction for all the 5 images but it gives similar kind of guess based on the image.

The model is not that certain with the prediction with only image-1, image-4 and image-5 having more than 50%. And among that only image-1 being around 80%.