**Convolutional Neural Networks: Application**

Welcome to Course 4's second assignment! In this notebook, you will:

* Create a mood classifer using the TF Keras Sequential API
* Build a ConvNet to identify sign language digits using the TF Keras Functional API

**After this assignment you will be able to:**

* Build and train a ConvNet in TensorFlow for a **binary** classification problem
* Build and train a ConvNet in TensorFlow for a **multiclass** classification problem
* Explain different use cases for the Sequential and Functional APIs

To complete this assignment, you should already be familiar with TensorFlow. If you are not, please refer back to the **TensorFlow Tutorial** of the third week of Course 2 ("**Improving deep neural networks**").

**Important Note on Submission to the AutoGrader**

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any *extra* print statement(s) in the assignment.
2. You have not added any *extra* code cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating *extra* variables.

If you do any of the following, you will get something like, Grader not found (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these [instructions](https://www.coursera.org/learn/convolutional-neural-networks/supplement/DS4yP/h-ow-to-refresh-your-workspace).

**Table of Contents**

* [1 - Packages](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#1)
  + [1.1 - Load the Data and Split the Data into Train/Test Sets](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#1-1)
* [2 - Layers in TF Keras](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#2)
* [3 - The Sequential API](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#3)
  + [3.1 - Create the Sequential Model](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#3-1)
    - [Exercise 1 - happyModel](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#ex-1)
  + [3.2 - Train and Evaluate the Model](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#3-2)
* [4 - The Functional API](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#4)
  + [4.1 - Load the SIGNS Dataset](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#4-1)
  + [4.2 - Split the Data into Train/Test Sets](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#4-2)
  + [4.3 - Forward Propagation](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#4-3)
    - [Exercise 2 - convolutional\_model](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#ex-2)
  + [4.4 - Train the Model](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#4-4)
* [5 - History Object](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#5)
* [6 - Bibliography](https://lyowynvz.labs.coursera.org/notebooks/release/W1A2/Convolution_model_Application.ipynb#6)

**1 - Packages**

As usual, begin by loading in the packages.

In [3]:

**import** math

**import** numpy **as** np

**import** h5py

**import** matplotlib.pyplot **as** plt

**from** matplotlib.pyplot **import** imread

**import** scipy

**from** PIL **import** Image

**import** pandas **as** pd

**import** tensorflow **as** tf

**import** tensorflow.keras.layers **as** tfl

**from** tensorflow.python.framework **import** ops

**from** cnn\_utils **import** **\***

**from** test\_utils **import** summary, comparator

​

**%**matplotlib inline

np.random.seed(1)

**1.1 - Load the Data and Split the Data into Train/Test Sets**

You'll be using the Happy House dataset for this part of the assignment, which contains images of peoples' faces. Your task will be to build a ConvNet that determines whether the people in the images are smiling or not -- because they only get to enter the house if they're smiling!

In [4]:

X\_train\_orig, Y\_train\_orig, X\_test\_orig, Y\_test\_orig, classes **=** load\_happy\_dataset()

​

*# Normalize image vectors*

X\_train **=** X\_train\_orig**/**255.

X\_test **=** X\_test\_orig**/**255.

​

*# Reshape*

Y\_train **=** Y\_train\_orig.T

Y\_test **=** Y\_test\_orig.T

​

print ("number of training examples = " **+** str(X\_train.shape[0]))

print ("number of test examples = " **+** str(X\_test.shape[0]))

print ("X\_train shape: " **+** str(X\_train.shape))

print ("Y\_train shape: " **+** str(Y\_train.shape))

print ("X\_test shape: " **+** str(X\_test.shape))

print ("Y\_test shape: " **+** str(Y\_test.shape))

number of training examples = 600

number of test examples = 150

X\_train shape: (600, 64, 64, 3)

Y\_train shape: (600, 1)

X\_test shape: (150, 64, 64, 3)

Y\_test shape: (150, 1)

You can display the images contained in the dataset. Images are **64x64** pixels in RGB format (3 channels).

In [5]:

index **=** 7

plt.imshow(X\_train\_orig[index]) *#display sample training image*

plt.show()

A person taking a selfie

Description automatically generated

**2 - Layers in TF Keras**

In the previous assignment, you created layers manually in numpy. In TF Keras, you don't have to write code directly to create layers. Rather, TF Keras has pre-defined layers you can use.

When you create a layer in TF Keras, you are creating a function that takes some input and transforms it into an output you can reuse later. Nice and easy!

**3 - The Sequential API**

In the previous assignment, you built helper functions using numpy to understand the mechanics behind convolutional neural networks. Most practical applications of deep learning today are built using programming frameworks, which have many built-in functions you can simply call. Keras is a high-level abstraction built on top of TensorFlow, which allows for even more simplified and optimized model creation and training.

For the first part of this assignment, you'll create a model using TF Keras' Sequential API, which allows you to build layer by layer, and is ideal for building models where each layer has **exactly one** input tensor and **one** output tensor.

As you'll see, using the Sequential API is simple and straightforward, but is only appropriate for simpler, more straightforward tasks. Later in this notebook you'll spend some time building with a more flexible, powerful alternative: the Functional API.

**3.1 - Create the Sequential Model**

As mentioned earlier, the TensorFlow Keras Sequential API can be used to build simple models with layer operations that proceed in a sequential order.

You can also add layers incrementally to a Sequential model with the .add() method, or remove them using the .pop() method, much like you would in a regular Python list.

Actually, you can think of a Sequential model as behaving like a list of layers. Like Python lists, Sequential layers are ordered, and the order in which they are specified matters. If your model is non-linear or contains layers with multiple inputs or outputs, a Sequential model wouldn't be the right choice!

For any layer construction in Keras, you'll need to specify the input shape in advance. This is because in Keras, the shape of the weights is based on the shape of the inputs. The weights are only created when the model first sees some input data. Sequential models can be created by passing a list of layers to the Sequential constructor, like you will do in the next assignment.

**Exercise 1 - happyModel**

Implement the happyModel function below to build the following model: ZEROPAD2D -> CONV2D -> BATCHNORM -> RELU -> MAXPOOL -> FLATTEN -> DENSE. Take help from [tf.keras.layers](https://www.tensorflow.org/api_docs/python/tf/keras/layers)

Also, plug in the following parameters for all the steps:

* [ZeroPadding2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/ZeroPadding2D): padding 3, input shape 64 x 64 x 3
* [Conv2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D): Use 32 7x7 filters, stride 1
* [BatchNormalization](https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization): for axis 3
* [ReLU](https://www.tensorflow.org/api_docs/python/tf/keras/layers/ReLU)
* [MaxPool2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D): Using default parameters
* [Flatten](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Flatten) the previous output.
* Fully-connected ([Dense](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)) layer: Apply a fully connected layer with 1 neuron and a sigmoid activation.

**Hint:**

Use **tfl** as shorthand for **tensorflow.keras.layers**

In [6]:

*# GRADED FUNCTION: happyModel*

​

**def** happyModel():

"""

Implements the forward propagation for the binary classification model:

ZEROPAD2D -> CONV2D -> BATCHNORM -> RELU -> MAXPOOL -> FLATTEN -> DENSE

Note that for simplicity and grading purposes, you'll hard-code all the values

such as the stride and kernel (filter) sizes.

Normally, functions should take these values as function parameters.

Arguments:

None

​

Returns:

model -- TF Keras model (object containing the information for the entire training process)

"""

model **=** tf.keras.Sequential([

*## ZeroPadding2D with padding 3, input shape of 64 x 64 x 3*

tfl.ZeroPadding2D(padding**=**(3,3),input\_shape**=**(64, 64, 3), data\_format**=**"channels\_last"),

*## Conv2D with 32 7x7 filters and stride of 1*

tfl.Conv2D(32, 7, strides**=**(1, 1)),

*## BatchNormalization for axis 3*

tfl.BatchNormalization(axis**=**3),

*## ReLU*

tfl.ReLU(),

*## Max Pooling 2D with default parameters*

tfl.MaxPool2D(),

*## Flatten layer*

tfl.Flatten(),

*## Dense layer with 1 unit for output & 'sigmoid' activation*

tfl.Dense(1, activation**=** "sigmoid")

​

])

**return** model

In [7]:

happy\_model **=** happyModel()

*# Print a summary for each layer*

**for** layer **in** summary(happy\_model):

print(layer)

output **=** [['ZeroPadding2D', (**None**, 70, 70, 3), 0, ((3, 3), (3, 3))],

['Conv2D', (**None**, 64, 64, 32), 4736, 'valid', 'linear', 'GlorotUniform'],

['BatchNormalization', (**None**, 64, 64, 32), 128],

['ReLU', (**None**, 64, 64, 32), 0],

['MaxPooling2D', (**None**, 32, 32, 32), 0, (2, 2), (2, 2), 'valid'],

['Flatten', (**None**, 32768), 0],

['Dense', (**None**, 1), 32769, 'sigmoid']]

comparator(summary(happy\_model), output)

['ZeroPadding2D', (None, 70, 70, 3), 0, ((3, 3), (3, 3))]

['Conv2D', (None, 64, 64, 32), 4736, 'valid', 'linear', 'GlorotUniform']

['BatchNormalization', (None, 64, 64, 32), 128]

['ReLU', (None, 64, 64, 32), 0]

['MaxPooling2D', (None, 32, 32, 32), 0, (2, 2), (2, 2), 'valid']

['Flatten', (None, 32768), 0]

['Dense', (None, 1), 32769, 'sigmoid']

All tests passed!

Now that your model is created, you can compile it for training with an optimizer and loss of your choice. When the string accuracy is specified as a metric, the type of accuracy used will be automatically converted based on the loss function used. This is one of the many optimizations built into TensorFlow that make your life easier! If you'd like to read more on how the compiler operates, check the docs [here](https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile).

In [8]:

happy\_model.compile(optimizer**=**'adam',

loss**=**'binary\_crossentropy',

metrics**=**['accuracy'])

It's time to check your model's parameters with the .summary() method. This will display the types of layers you have, the shape of the outputs, and how many parameters are in each layer.

In [9]:

happy\_model.summary()

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

zero\_padding2d (ZeroPadding2 (None, 70, 70, 3) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d (Conv2D) (None, 64, 64, 32) 4736

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization (BatchNo (None, 64, 64, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

re\_lu (ReLU) (None, 64, 64, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d (MaxPooling2D) (None, 32, 32, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 32768) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 1) 32769

=================================================================

Total params: 37,633

Trainable params: 37,569

Non-trainable params: 64

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3.2 - Train and Evaluate the Model**

After creating the model, compiling it with your choice of optimizer and loss function, and doing a sanity check on its contents, you are now ready to build!

Simply call .fit() to train. That's it! No need for mini-batching, saving, or complex backpropagation computations. That's all been done for you, as you're using a TensorFlow dataset with the batches specified already. You do have the option to specify epoch number or minibatch size if you like (for example, in the case of an un-batched dataset).

In [10]:

happy\_model.fit(X\_train, Y\_train, epochs**=**10, batch\_size**=**16)

Epoch 1/10

38/38 [==============================] - 4s 103ms/step - loss: 0.9058 - accuracy: 0.7550

Epoch 2/10

38/38 [==============================] - 4s 97ms/step - loss: 0.2816 - accuracy: 0.8967

Epoch 3/10

38/38 [==============================] - 4s 100ms/step - loss: 0.1572 - accuracy: 0.9317

Epoch 4/10

38/38 [==============================] - 4s 95ms/step - loss: 0.1506 - accuracy: 0.9300

Epoch 5/10

38/38 [==============================] - 4s 97ms/step - loss: 0.1369 - accuracy: 0.9467

Epoch 6/10

38/38 [==============================] - 4s 100ms/step - loss: 0.1106 - accuracy: 0.9700

Epoch 7/10

38/38 [==============================] - 4s 95ms/step - loss: 0.1298 - accuracy: 0.9650

Epoch 8/10

38/38 [==============================] - 4s 97ms/step - loss: 0.0888 - accuracy: 0.9717

Epoch 9/10

38/38 [==============================] - 4s 98ms/step - loss: 0.1204 - accuracy: 0.9500

Epoch 10/10

38/38 [==============================] - 4s 97ms/step - loss: 0.0462 - accuracy: 0.9800

Out[10]:

<tensorflow.python.keras.callbacks.History at 0x7f89e6909650>

After that completes, just use .evaluate() to evaluate against your test set. This function will print the value of the loss function and the performance metrics specified during the compilation of the model. In this case, the binary\_crossentropy and the accuracy respectively.

In [11]:

happy\_model.evaluate(X\_test, Y\_test)

5/5 [==============================] - 0s 30ms/step - loss: 0.1554 - accuracy: 0.9267

Out[11]:

[0.1554018259048462, 0.9266666769981384]

Easy, right? But what if you need to build a model with shared layers, branches, or multiple inputs and outputs? This is where Sequential, with its beautifully simple yet limited functionality, won't be able to help you.

Next up: Enter the Functional API, your slightly more complex, highly flexible friend.

**4 - The Functional API**

Welcome to the second half of the assignment, where you'll use Keras' flexible [Functional API](https://www.tensorflow.org/guide/keras/functional) to build a ConvNet that can differentiate between 6 sign language digits.

The Functional API can handle models with non-linear topology, shared layers, as well as layers with multiple inputs or outputs. Imagine that, where the Sequential API requires the model to move in a linear fashion through its layers, the Functional API allows much more flexibility. Where Sequential is a straight line, a Functional model is a graph, where the nodes of the layers can connect in many more ways than one.

In the visual example below, the one possible direction of the movement Sequential model is shown in contrast to a skip connection, which is just one of the many ways a Functional model can be constructed. A skip connection, as you might have guessed, skips some layer in the network and feeds the output to a later layer in the network. Don't worry, you'll be spending more time with skip connections very soon!

**4.1 - Load the SIGNS Dataset**

As a reminder, the SIGNS dataset is a collection of 6 signs representing numbers from 0 to 5.

In [12]:

*# Loading the data (signs)*

X\_train\_orig, Y\_train\_orig, X\_test\_orig, Y\_test\_orig, classes **=** load\_signs\_dataset()

The next cell will show you an example of a labelled image in the dataset. Feel free to change the value of index below and re-run to see different examples.

In [13]:

*# Example of an image from the dataset*

index **=** 9

plt.imshow(X\_train\_orig[index])

print ("y = " **+** str(np.squeeze(Y\_train\_orig[:, index])))

y = 4

A picture containing text, handwear, band-aid

Description automatically generated

**4.2 - Split the Data into Train/Test Sets**

In Course 2, you built a fully-connected network for this dataset. But since this is an image dataset, it is more natural to apply a ConvNet to it.

To get started, let's examine the shapes of your data.

In [14]:

X\_train **=** X\_train\_orig**/**255.

X\_test **=** X\_test\_orig**/**255.

Y\_train **=** convert\_to\_one\_hot(Y\_train\_orig, 6).T

Y\_test **=** convert\_to\_one\_hot(Y\_test\_orig, 6).T

print ("number of training examples = " **+** str(X\_train.shape[0]))

print ("number of test examples = " **+** str(X\_test.shape[0]))

print ("X\_train shape: " **+** str(X\_train.shape))

print ("Y\_train shape: " **+** str(Y\_train.shape))

print ("X\_test shape: " **+** str(X\_test.shape))

print ("Y\_test shape: " **+** str(Y\_test.shape))

number of training examples = 1080

number of test examples = 120

X\_train shape: (1080, 64, 64, 3)

Y\_train shape: (1080, 6)

X\_test shape: (120, 64, 64, 3)

Y\_test shape: (120, 6)

**4.3 - Forward Propagation**

In TensorFlow, there are built-in functions that implement the convolution steps for you. By now, you should be familiar with how TensorFlow builds computational graphs. In the [Functional API](https://www.tensorflow.org/guide/keras/functional), you create a graph of layers. This is what allows such great flexibility.

However, the following model could also be defined using the Sequential API since the information flow is on a single line. But don't deviate. What we want you to learn is to use the functional API.

Begin building your graph of layers by creating an input node that functions as a callable object:

* **input\_img = tf.keras.Input(shape=input\_shape):**

Then, create a new node in the graph of layers by calling a layer on the input\_img object:

* **tf.keras.layers.Conv2D(filters= ... , kernel\_size= ... , padding='same')(input\_img):** Read the full documentation on [Conv2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D).
* **tf.keras.layers.MaxPool2D(pool\_size=(f, f), strides=(s, s), padding='same'):** MaxPool2D() downsamples your input using a window of size (f, f) and strides of size (s, s) to carry out max pooling over each window. For max pooling, you usually operate on a single example at a time and a single channel at a time. Read the full documentation on [MaxPool2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D).
* **tf.keras.layers.ReLU():** computes the elementwise ReLU of Z (which can be any shape). You can read the full documentation on [ReLU](https://www.tensorflow.org/api_docs/python/tf/keras/layers/ReLU).
* **tf.keras.layers.Flatten()**: given a tensor "P", this function takes each training (or test) example in the batch and flattens it into a 1D vector.
  + If a tensor P has the shape (batch\_size,h,w,c), it returns a flattened tensor with shape (batch\_size, k), where 𝑘=ℎ×𝑤×𝑐k=h×w×c. "k" equals the product of all the dimension sizes other than the first dimension.
  + For example, given a tensor with dimensions [100, 2, 3, 4], it flattens the tensor to be of shape [100, 24], where 24 = 2 \* 3 \* 4. You can read the full documentation on [Flatten](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Flatten).
* **tf.keras.layers.Dense(units= ... , activation='softmax')(F):** given the flattened input F, it returns the output computed using a fully connected layer. You can read the full documentation on [Dense](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense).

In the last function above (tf.keras.layers.Dense()), the fully connected layer automatically initializes weights in the graph and keeps on training them as you train the model. Hence, you did not need to initialize those weights when initializing the parameters.

Lastly, before creating the model, you'll need to define the output using the last of the function's compositions (in this example, a Dense layer):

* **outputs = tf.keras.layers.Dense(units=6, activation='softmax')(F)**

**Window, kernel, filter, pool**

The words "kernel" and "filter" are used to refer to the same thing. The word "filter" accounts for the amount of "kernels" that will be used in a single convolution layer. "Pool" is the name of the operation that takes the max or average value of the kernels.

This is why the parameter pool\_size refers to kernel\_size, and you use (f,f) to refer to the filter size.

Pool size and kernel size refer to the same thing in different objects - They refer to the shape of the window where the operation takes place.

**Exercise 2 - convolutional\_model**

Implement the convolutional\_model function below to build the following model: CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> DENSE. Use the functions above!

Also, plug in the following parameters for all the steps:

* [Conv2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D): Use 8 4 by 4 filters, stride 1, padding is "SAME"
* [ReLU](https://www.tensorflow.org/api_docs/python/tf/keras/layers/ReLU)
* [MaxPool2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D): Use an 8 by 8 filter size and an 8 by 8 stride, padding is "SAME"
* **Conv2D**: Use 16 2 by 2 filters, stride 1, padding is "SAME"
* **ReLU**
* **MaxPool2D**: Use a 4 by 4 filter size and a 4 by 4 stride, padding is "SAME"
* [Flatten](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Flatten) the previous output.
* Fully-connected ([Dense](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)) layer: Apply a fully connected layer with 6 neurons and a softmax activation.

In [31]:

*# GRADED FUNCTION: convolutional\_model*

​

**def** convolutional\_model(input\_shape):

"""

Implements the forward propagation for the model:

CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> DENSE

Note that for simplicity and grading purposes, you'll hard-code some values

such as the stride and kernel (filter) sizes.

Normally, functions should take these values as function parameters.

Arguments:

input\_img -- input dataset, of shape (input\_shape)

​

Returns:

model -- TF Keras model (object containing the information for the entire training process)

"""

​

input\_img **=** tf.keras.Input(shape**=**input\_shape)

*## CONV2D: 8 filters 4x4, stride of 1, padding 'SAME'*

Z1 **=** tfl.Conv2D(filters**=** 8 , kernel\_size**=** (4,4) , strides**=**(1, 1) ,padding**=**'same')(input\_img)

*## RELU*

A1 **=** tfl.ReLU()(Z1)

*## MAXPOOL: window 8x8, stride 8, padding 'SAME'*

P1 **=** tfl.MaxPool2D(pool\_size**=**(8, 8), strides**=**(8, 8), padding**=**'same')(A1)

*## CONV2D: 16 filters 2x2, stride 1, padding 'SAME'*

Z2 **=** tfl.Conv2D(filters**=** 16 , kernel\_size**=** (2,2) , strides**=**(1, 1) ,padding**=**'same')(P1)

*## RELU*

A2 **=** tfl.ReLU()(Z2)

*## MAXPOOL: window 4x4, stride 4, padding 'SAME'*

P2 **=** tfl.MaxPool2D(pool\_size**=**(4, 4), strides**=**(4, 4), padding**=**'same')(A2)

*## FLATTEN*

F **=** tfl.Flatten()(P2)

*## Dense layer*

*## 6 neurons in output layer. Hint: one of the arguments should be "activation='softmax'"*

outputs **=** tfl.Dense(units**=** 6 , activation**=**'softmax')(F)

*# YOUR CODE STARTS HERE*

*# YOUR CODE ENDS HERE*

model **=** tf.keras.Model(inputs**=**input\_img, outputs**=**outputs)

**return** model

In [32]:

conv\_model **=** convolutional\_model((64, 64, 3))

conv\_model.compile(optimizer**=**'adam',

loss**=**'categorical\_crossentropy',

metrics**=**['accuracy'])

conv\_model.summary()

output **=** [['InputLayer', [(**None**, 64, 64, 3)], 0],

['Conv2D', (**None**, 64, 64, 8), 392, 'same', 'linear', 'GlorotUniform'],

['ReLU', (**None**, 64, 64, 8), 0],

['MaxPooling2D', (**None**, 8, 8, 8), 0, (8, 8), (8, 8), 'same'],

['Conv2D', (**None**, 8, 8, 16), 528, 'same', 'linear', 'GlorotUniform'],

['ReLU', (**None**, 8, 8, 16), 0],

['MaxPooling2D', (**None**, 2, 2, 16), 0, (4, 4), (4, 4), 'same'],

['Flatten', (**None**, 64), 0],

['Dense', (**None**, 6), 390, 'softmax']]

comparator(summary(conv\_model), output)

Model: "functional\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_9 (InputLayer) [(None, 64, 64, 3)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_15 (Conv2D) (None, 64, 64, 8) 392

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

re\_lu\_15 (ReLU) (None, 64, 64, 8) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_15 (MaxPooling (None, 8, 8, 8) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_16 (Conv2D) (None, 8, 8, 16) 528

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

re\_lu\_16 (ReLU) (None, 8, 8, 16) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_16 (MaxPooling (None, 2, 2, 16) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_8 (Flatten) (None, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_8 (Dense) (None, 6) 390

=================================================================

Total params: 1,310

Trainable params: 1,310

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

All tests passed!

Both the Sequential and Functional APIs return a TF Keras model object. The only difference is how inputs are handled inside the object model!

**4.4 - Train the Model**

In [33]:

train\_dataset **=** tf.data.Dataset.from\_tensor\_slices((X\_train, Y\_train)).batch(64)

test\_dataset **=** tf.data.Dataset.from\_tensor\_slices((X\_test, Y\_test)).batch(64)

history **=** conv\_model.fit(train\_dataset, epochs**=**100, validation\_data**=**test\_dataset)

Epoch 1/100

17/17 [==============================] - 2s 106ms/step - loss: 1.7947 - accuracy: 0.1722 - val\_loss: 1.7898 - val\_accuracy: 0.2417

Epoch 2/100

17/17 [==============================] - 2s 106ms/step - loss: 1.7882 - accuracy: 0.2444 - val\_loss: 1.7865 - val\_accuracy: 0.2500

Epoch 3/100

17/17 [==============================] - 2s 106ms/step - loss: 1.7845 - accuracy: 0.2556 - val\_loss: 1.7840 - val\_accuracy: 0.2750

Epoch 4/100

17/17 [==============================] - 2s 106ms/step - loss: 1.7797 - accuracy: 0.2944 - val\_loss: 1.7801 - val\_accuracy: 0.2583

Epoch 5/100

17/17 [==============================] - 2s 101ms/step - loss: 1.7733 - accuracy: 0.2907 - val\_loss: 1.7735 - val\_accuracy: 0.3167

Epoch 6/100

17/17 [==============================] - 2s 102ms/step - loss: 1.7639 - accuracy: 0.2898 - val\_loss: 1.7655 - val\_accuracy: 0.2667

Epoch 7/100

17/17 [==============================] - 2s 101ms/step - loss: 1.7523 - accuracy: 0.2769 - val\_loss: 1.7536 - val\_accuracy: 0.3250

Epoch 8/100

17/17 [==============================] - 2s 106ms/step - loss: 1.7355 - accuracy: 0.3157 - val\_loss: 1.7375 - val\_accuracy: 0.3417

Epoch 9/100

17/17 [==============================] - 2s 106ms/step - loss: 1.7134 - accuracy: 0.3556 - val\_loss: 1.7154 - val\_accuracy: 0.3667

Epoch 10/100

17/17 [==============================] - 2s 106ms/step - loss: 1.6852 - accuracy: 0.3981 - val\_loss: 1.6875 - val\_accuracy: 0.4417

Epoch 11/100

17/17 [==============================] - 2s 106ms/step - loss: 1.6503 - accuracy: 0.4435 - val\_loss: 1.6517 - val\_accuracy: 0.4667

Epoch 12/100

17/17 [==============================] - 2s 106ms/step - loss: 1.6070 - accuracy: 0.4852 - val\_loss: 1.6083 - val\_accuracy: 0.4917

Epoch 13/100

17/17 [==============================] - 2s 106ms/step - loss: 1.5569 - accuracy: 0.5056 - val\_loss: 1.5583 - val\_accuracy: 0.5000

Epoch 14/100

17/17 [==============================] - 2s 106ms/step - loss: 1.5027 - accuracy: 0.5278 - val\_loss: 1.5059 - val\_accuracy: 0.5417

Epoch 15/100

17/17 [==============================] - 2s 106ms/step - loss: 1.4487 - accuracy: 0.5444 - val\_loss: 1.4532 - val\_accuracy: 0.5333

Epoch 16/100

17/17 [==============================] - 2s 106ms/step - loss: 1.3974 - accuracy: 0.5537 - val\_loss: 1.4038 - val\_accuracy: 0.5417

Epoch 17/100

17/17 [==============================] - 2s 101ms/step - loss: 1.3496 - accuracy: 0.5648 - val\_loss: 1.3586 - val\_accuracy: 0.5500

Epoch 18/100

17/17 [==============================] - 2s 106ms/step - loss: 1.3060 - accuracy: 0.5741 - val\_loss: 1.3146 - val\_accuracy: 0.5417

Epoch 19/100

17/17 [==============================] - 2s 101ms/step - loss: 1.2664 - accuracy: 0.5806 - val\_loss: 1.2717 - val\_accuracy: 0.5500

Epoch 20/100

17/17 [==============================] - 2s 106ms/step - loss: 1.2279 - accuracy: 0.5926 - val\_loss: 1.2312 - val\_accuracy: 0.5833

Epoch 21/100

17/17 [==============================] - 2s 106ms/step - loss: 1.1917 - accuracy: 0.6148 - val\_loss: 1.1925 - val\_accuracy: 0.6083

Epoch 22/100

17/17 [==============================] - 2s 106ms/step - loss: 1.1564 - accuracy: 0.6259 - val\_loss: 1.1560 - val\_accuracy: 0.6083

Epoch 23/100

17/17 [==============================] - 2s 101ms/step - loss: 1.1236 - accuracy: 0.6352 - val\_loss: 1.1210 - val\_accuracy: 0.6167

Epoch 24/100

17/17 [==============================] - 2s 106ms/step - loss: 1.0916 - accuracy: 0.6500 - val\_loss: 1.0874 - val\_accuracy: 0.6250

Epoch 25/100

17/17 [==============================] - 2s 106ms/step - loss: 1.0606 - accuracy: 0.6611 - val\_loss: 1.0541 - val\_accuracy: 0.6250

Epoch 26/100

17/17 [==============================] - 2s 106ms/step - loss: 1.0294 - accuracy: 0.6676 - val\_loss: 1.0243 - val\_accuracy: 0.6500

Epoch 27/100

17/17 [==============================] - 2s 101ms/step - loss: 1.0009 - accuracy: 0.6852 - val\_loss: 0.9946 - val\_accuracy: 0.6667

Epoch 28/100

17/17 [==============================] - 2s 106ms/step - loss: 0.9733 - accuracy: 0.7000 - val\_loss: 0.9678 - val\_accuracy: 0.6750

Epoch 29/100

17/17 [==============================] - 2s 105ms/step - loss: 0.9477 - accuracy: 0.7093 - val\_loss: 0.9418 - val\_accuracy: 0.6833

Epoch 30/100

17/17 [==============================] - 2s 105ms/step - loss: 0.9234 - accuracy: 0.7194 - val\_loss: 0.9171 - val\_accuracy: 0.6833

Epoch 31/100

17/17 [==============================] - 2s 106ms/step - loss: 0.8997 - accuracy: 0.7296 - val\_loss: 0.8945 - val\_accuracy: 0.6917

Epoch 32/100

17/17 [==============================] - 2s 106ms/step - loss: 0.8777 - accuracy: 0.7352 - val\_loss: 0.8733 - val\_accuracy: 0.6917

Epoch 33/100

17/17 [==============================] - 2s 106ms/step - loss: 0.8559 - accuracy: 0.7389 - val\_loss: 0.8536 - val\_accuracy: 0.7083

Epoch 34/100

17/17 [==============================] - 2s 106ms/step - loss: 0.8355 - accuracy: 0.7491 - val\_loss: 0.8336 - val\_accuracy: 0.7250

Epoch 35/100

17/17 [==============================] - 2s 106ms/step - loss: 0.8170 - accuracy: 0.7546 - val\_loss: 0.8161 - val\_accuracy: 0.7417

Epoch 36/100

17/17 [==============================] - 2s 106ms/step - loss: 0.7990 - accuracy: 0.7574 - val\_loss: 0.7988 - val\_accuracy: 0.7417

Epoch 37/100

17/17 [==============================] - 2s 106ms/step - loss: 0.7822 - accuracy: 0.7630 - val\_loss: 0.7823 - val\_accuracy: 0.7333

Epoch 38/100

17/17 [==============================] - 2s 106ms/step - loss: 0.7662 - accuracy: 0.7667 - val\_loss: 0.7670 - val\_accuracy: 0.7417

Epoch 39/100

17/17 [==============================] - 2s 106ms/step - loss: 0.7509 - accuracy: 0.7722 - val\_loss: 0.7519 - val\_accuracy: 0.7583

Epoch 40/100

17/17 [==============================] - 2s 106ms/step - loss: 0.7363 - accuracy: 0.7722 - val\_loss: 0.7380 - val\_accuracy: 0.7667

Epoch 41/100

17/17 [==============================] - 2s 106ms/step - loss: 0.7222 - accuracy: 0.7731 - val\_loss: 0.7259 - val\_accuracy: 0.7750

Epoch 42/100

17/17 [==============================] - 2s 100ms/step - loss: 0.7091 - accuracy: 0.7750 - val\_loss: 0.7146 - val\_accuracy: 0.7667

Epoch 43/100

17/17 [==============================] - 2s 106ms/step - loss: 0.6964 - accuracy: 0.7759 - val\_loss: 0.7036 - val\_accuracy: 0.7833

Epoch 44/100

17/17 [==============================] - 2s 106ms/step - loss: 0.6843 - accuracy: 0.7796 - val\_loss: 0.6931 - val\_accuracy: 0.7833

Epoch 45/100

17/17 [==============================] - 2s 106ms/step - loss: 0.6733 - accuracy: 0.7824 - val\_loss: 0.6830 - val\_accuracy: 0.7917

Epoch 46/100

17/17 [==============================] - 2s 106ms/step - loss: 0.6625 - accuracy: 0.7861 - val\_loss: 0.6734 - val\_accuracy: 0.7917

Epoch 47/100

17/17 [==============================] - 2s 101ms/step - loss: 0.6523 - accuracy: 0.7926 - val\_loss: 0.6641 - val\_accuracy: 0.7917

Epoch 48/100

17/17 [==============================] - 2s 100ms/step - loss: 0.6424 - accuracy: 0.7963 - val\_loss: 0.6554 - val\_accuracy: 0.7917

Epoch 49/100

17/17 [==============================] - 2s 101ms/step - loss: 0.6328 - accuracy: 0.7981 - val\_loss: 0.6466 - val\_accuracy: 0.8000

Epoch 50/100

17/17 [==============================] - 2s 101ms/step - loss: 0.6234 - accuracy: 0.8028 - val\_loss: 0.6385 - val\_accuracy: 0.8000

Epoch 51/100

17/17 [==============================] - 2s 106ms/step - loss: 0.6145 - accuracy: 0.8037 - val\_loss: 0.6303 - val\_accuracy: 0.8000

Epoch 52/100

17/17 [==============================] - 2s 106ms/step - loss: 0.6060 - accuracy: 0.8056 - val\_loss: 0.6224 - val\_accuracy: 0.8000

Epoch 53/100

17/17 [==============================] - 2s 101ms/step - loss: 0.5977 - accuracy: 0.8074 - val\_loss: 0.6148 - val\_accuracy: 0.8000

Epoch 54/100

17/17 [==============================] - 2s 101ms/step - loss: 0.5897 - accuracy: 0.8102 - val\_loss: 0.6073 - val\_accuracy: 0.8000

Epoch 55/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5817 - accuracy: 0.8139 - val\_loss: 0.6003 - val\_accuracy: 0.8000

Epoch 56/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5742 - accuracy: 0.8157 - val\_loss: 0.5936 - val\_accuracy: 0.8083

Epoch 57/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5667 - accuracy: 0.8176 - val\_loss: 0.5870 - val\_accuracy: 0.8167

Epoch 58/100

17/17 [==============================] - 2s 101ms/step - loss: 0.5596 - accuracy: 0.8185 - val\_loss: 0.5801 - val\_accuracy: 0.8167

Epoch 59/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5528 - accuracy: 0.8204 - val\_loss: 0.5742 - val\_accuracy: 0.8167

Epoch 60/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5460 - accuracy: 0.8194 - val\_loss: 0.5680 - val\_accuracy: 0.8167

Epoch 61/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5392 - accuracy: 0.8222 - val\_loss: 0.5620 - val\_accuracy: 0.8167

Epoch 62/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5327 - accuracy: 0.8241 - val\_loss: 0.5564 - val\_accuracy: 0.8167

Epoch 63/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5265 - accuracy: 0.8250 - val\_loss: 0.5508 - val\_accuracy: 0.8167

Epoch 64/100

17/17 [==============================] - 2s 101ms/step - loss: 0.5206 - accuracy: 0.8259 - val\_loss: 0.5457 - val\_accuracy: 0.8333

Epoch 65/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5146 - accuracy: 0.8296 - val\_loss: 0.5403 - val\_accuracy: 0.8333

Epoch 66/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5086 - accuracy: 0.8306 - val\_loss: 0.5348 - val\_accuracy: 0.8333

Epoch 67/100

17/17 [==============================] - 2s 106ms/step - loss: 0.5031 - accuracy: 0.8306 - val\_loss: 0.5297 - val\_accuracy: 0.8417

Epoch 68/100

17/17 [==============================] - 2s 101ms/step - loss: 0.4978 - accuracy: 0.8343 - val\_loss: 0.5245 - val\_accuracy: 0.8417

Epoch 69/100

17/17 [==============================] - 2s 101ms/step - loss: 0.4926 - accuracy: 0.8361 - val\_loss: 0.5193 - val\_accuracy: 0.8417

Epoch 70/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4875 - accuracy: 0.8389 - val\_loss: 0.5145 - val\_accuracy: 0.8417

Epoch 71/100

17/17 [==============================] - 2s 100ms/step - loss: 0.4825 - accuracy: 0.8407 - val\_loss: 0.5098 - val\_accuracy: 0.8417

Epoch 72/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4777 - accuracy: 0.8417 - val\_loss: 0.5053 - val\_accuracy: 0.8417

Epoch 73/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4726 - accuracy: 0.8426 - val\_loss: 0.5007 - val\_accuracy: 0.8333

Epoch 74/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4680 - accuracy: 0.8472 - val\_loss: 0.4968 - val\_accuracy: 0.8333

Epoch 75/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4633 - accuracy: 0.8491 - val\_loss: 0.4927 - val\_accuracy: 0.8333

Epoch 76/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4589 - accuracy: 0.8528 - val\_loss: 0.4885 - val\_accuracy: 0.8333

Epoch 77/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4543 - accuracy: 0.8519 - val\_loss: 0.4845 - val\_accuracy: 0.8333

Epoch 78/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4504 - accuracy: 0.8556 - val\_loss: 0.4813 - val\_accuracy: 0.8333

Epoch 79/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4462 - accuracy: 0.8593 - val\_loss: 0.4780 - val\_accuracy: 0.8333

Epoch 80/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4420 - accuracy: 0.8593 - val\_loss: 0.4745 - val\_accuracy: 0.8333

Epoch 81/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4380 - accuracy: 0.8620 - val\_loss: 0.4713 - val\_accuracy: 0.8333

Epoch 82/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4337 - accuracy: 0.8630 - val\_loss: 0.4679 - val\_accuracy: 0.8333

Epoch 83/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4299 - accuracy: 0.8639 - val\_loss: 0.4649 - val\_accuracy: 0.8417

Epoch 84/100

17/17 [==============================] - 2s 101ms/step - loss: 0.4261 - accuracy: 0.8667 - val\_loss: 0.4618 - val\_accuracy: 0.8417

Epoch 85/100

17/17 [==============================] - 2s 100ms/step - loss: 0.4223 - accuracy: 0.8685 - val\_loss: 0.4588 - val\_accuracy: 0.8417

Epoch 86/100

17/17 [==============================] - 2s 101ms/step - loss: 0.4188 - accuracy: 0.8694 - val\_loss: 0.4559 - val\_accuracy: 0.8500

Epoch 87/100

17/17 [==============================] - 2s 100ms/step - loss: 0.4150 - accuracy: 0.8704 - val\_loss: 0.4527 - val\_accuracy: 0.8500

Epoch 88/100

17/17 [==============================] - 2s 100ms/step - loss: 0.4113 - accuracy: 0.8750 - val\_loss: 0.4498 - val\_accuracy: 0.8500

Epoch 89/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4076 - accuracy: 0.8750 - val\_loss: 0.4468 - val\_accuracy: 0.8500

Epoch 90/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4042 - accuracy: 0.8759 - val\_loss: 0.4442 - val\_accuracy: 0.8500

Epoch 91/100

17/17 [==============================] - 2s 106ms/step - loss: 0.4008 - accuracy: 0.8769 - val\_loss: 0.4414 - val\_accuracy: 0.8500

Epoch 92/100

17/17 [==============================] - 2s 101ms/step - loss: 0.3976 - accuracy: 0.8787 - val\_loss: 0.4391 - val\_accuracy: 0.8417

Epoch 93/100

17/17 [==============================] - 2s 101ms/step - loss: 0.3944 - accuracy: 0.8796 - val\_loss: 0.4366 - val\_accuracy: 0.8417

Epoch 94/100

17/17 [==============================] - 2s 106ms/step - loss: 0.3912 - accuracy: 0.8815 - val\_loss: 0.4337 - val\_accuracy: 0.8417

Epoch 95/100

17/17 [==============================] - 2s 106ms/step - loss: 0.3879 - accuracy: 0.8833 - val\_loss: 0.4315 - val\_accuracy: 0.8417

Epoch 96/100

17/17 [==============================] - 2s 106ms/step - loss: 0.3847 - accuracy: 0.8843 - val\_loss: 0.4288 - val\_accuracy: 0.8417

Epoch 97/100

17/17 [==============================] - 2s 102ms/step - loss: 0.3815 - accuracy: 0.8843 - val\_loss: 0.4264 - val\_accuracy: 0.8417

Epoch 98/100

17/17 [==============================] - 2s 101ms/step - loss: 0.3783 - accuracy: 0.8861 - val\_loss: 0.4232 - val\_accuracy: 0.8500

Epoch 99/100

17/17 [==============================] - 2s 106ms/step - loss: 0.3754 - accuracy: 0.8843 - val\_loss: 0.4212 - val\_accuracy: 0.8500

Epoch 100/100

17/17 [==============================] - 2s 106ms/step - loss: 0.3723 - accuracy: 0.8870 - val\_loss: 0.4188 - val\_accuracy: 0.8500

**5 - History Object**

The history object is an output of the .fit() operation, and provides a record of all the loss and metric values in memory. It's stored as a dictionary that you can retrieve at history.history:

In [34]:

history.history

Out[34]:

{'loss': [1.7946889400482178,

1.7881649732589722,

1.7845070362091064,

1.7796906232833862,

1.773349642753601,

1.7638930082321167,

1.7522916793823242,

1.735516905784607,

1.7133692502975464,

1.685218095779419,

1.6502975225448608,

1.6070102453231812,

1.556915283203125,

1.5027192831039429,

1.4486514329910278,

1.3974014520645142,

1.3495779037475586,

1.306044340133667,

1.2663862705230713,

1.2278976440429688,

1.1916804313659668,

1.1564041376113892,

1.1235672235488892,

1.0915653705596924,

1.0606422424316406,

1.0294158458709717,

1.0008573532104492,

0.9733301401138306,

0.9476526379585266,

0.9234433174133301,

0.8997483849525452,

0.877663791179657,

0.8559218049049377,

0.8354747295379639,

0.8169988989830017,

0.7990480661392212,

0.7821758985519409,

0.766220211982727,

0.7508677244186401,

0.7363471984863281,

0.7221795320510864,

0.7090772986412048,

0.6964097023010254,

0.6842731833457947,

0.6733074188232422,

0.6624746322631836,

0.6523188352584839,

0.6424224376678467,

0.6327984929084778,

0.6233882904052734,

0.6144534349441528,

0.6060171127319336,

0.5977205038070679,

0.5897032022476196,

0.5817452073097229,

0.5742274522781372,

0.5666676759719849,

0.5595652461051941,

0.552792489528656,

0.5459935069084167,

0.5391766428947449,

0.5327129364013672,

0.5265257954597473,

0.5206204056739807,

0.5146375894546509,

0.5086463093757629,

0.5031352639198303,

0.4978078305721283,

0.4925864338874817,

0.4875045120716095,

0.4824801981449127,

0.4776742458343506,

0.47264066338539124,

0.4679695665836334,

0.4632687270641327,

0.4588874578475952,

0.4542861580848694,

0.4503754675388336,

0.4462467133998871,

0.442001610994339,

0.43796104192733765,

0.43374085426330566,

0.42994558811187744,

0.4260595142841339,

0.4222736358642578,

0.4187532365322113,

0.41502827405929565,

0.4112505316734314,

0.407640665769577,

0.404238224029541,

0.40077632665634155,

0.3976045548915863,

0.39443886280059814,

0.3911708891391754,

0.387939989566803,

0.38473373651504517,

0.3814917206764221,

0.37834081053733826,

0.37535524368286133,

0.3723459243774414],

'accuracy': [0.17222222685813904,

0.24444444477558136,

0.25555557012557983,

0.2944444417953491,

0.2907407283782959,

0.2898148000240326,

0.2768518626689911,

0.3157407343387604,

0.35555556416511536,

0.39814814925193787,

0.4435185194015503,

0.4851851761341095,

0.5055555701255798,

0.5277777910232544,

0.5444444417953491,

0.5537037253379822,

0.5648148059844971,

0.5740740895271301,

0.5805555582046509,

0.5925925970077515,

0.614814817905426,

0.6259258985519409,

0.635185182094574,

0.6499999761581421,

0.6611111164093018,

0.6675925850868225,

0.6851851940155029,

0.699999988079071,

0.7092592716217041,

0.7194444537162781,

0.729629635810852,

0.7351852059364319,

0.7388888597488403,

0.7490741014480591,

0.7546296119689941,

0.7574074268341064,

0.7629629373550415,

0.7666666507720947,

0.7722222208976746,

0.7722222208976746,

0.7731481194496155,

0.7749999761581421,

0.7759259343147278,

0.779629647731781,

0.7824074029922485,

0.7861111164093018,

0.7925925850868225,

0.7962962985038757,

0.7981481552124023,

0.8027777671813965,

0.8037037253379822,

0.8055555820465088,

0.8074073791503906,

0.8101851940155029,

0.8138889074325562,

0.8157407641410828,

0.8175926208496094,

0.8185185194015503,

0.8203703761100769,

0.8194444179534912,

0.8222222328186035,

0.8240740895271301,

0.824999988079071,

0.8259259462356567,

0.8296296000480652,

0.8305555582046509,

0.8305555582046509,

0.8342592716217041,

0.8361111283302307,

0.8388888835906982,

0.8407407402992249,

0.8416666388511658,

0.8425925970077515,

0.8472222089767456,

0.8490740656852722,

0.8527777791023254,

0.8518518805503845,

0.855555534362793,

0.8592592477798462,

0.8592592477798462,

0.8620370626449585,

0.8629629611968994,

0.8638888597488403,

0.8666666746139526,

0.8685185313224792,

0.8694444298744202,

0.8703703880310059,

0.875,

0.875,

0.8759258985519409,

0.8768518567085266,

0.8787037134170532,

0.8796296119689941,

0.8814814686775208,

0.8833333253860474,

0.8842592835426331,

0.8842592835426331,

0.8861111402511597,

0.8842592835426331,

0.8870370388031006],

'val\_loss': [1.78981614112854,

1.7864750623703003,

1.783994436264038,

1.7800647020339966,

1.7735041379928589,

1.7655240297317505,

1.7536358833312988,

1.7375472784042358,

1.7154399156570435,

1.6874632835388184,

1.6516594886779785,

1.6082708835601807,

1.5582587718963623,

1.5058751106262207,

1.453209638595581,

1.403815507888794,

1.3585736751556396,

1.3145594596862793,

1.2716881036758423,

1.2311854362487793,

1.1924842596054077,

1.155950903892517,

1.1209598779678345,

1.0874054431915283,

1.0541101694107056,

1.024253249168396,

0.9946177005767822,

0.9678164124488831,

0.9417986273765564,

0.9170636534690857,

0.8944897055625916,

0.8733333349227905,

0.8535568714141846,

0.8336466550827026,

0.816055178642273,

0.7988161444664001,

0.7822749614715576,

0.7670397162437439,

0.7518594264984131,

0.7379568815231323,

0.7258893251419067,

0.7145904302597046,

0.7035858035087585,

0.6930784583091736,

0.6830305457115173,

0.6734454035758972,

0.6641108393669128,

0.655369222164154,

0.6465709805488586,

0.6384814977645874,

0.6303239464759827,

0.6223634481430054,

0.6147642731666565,

0.6072757244110107,

0.6002810001373291,

0.5935718417167664,

0.5869768857955933,

0.5801206231117249,

0.5741985440254211,

0.568036675453186,

0.5620291233062744,

0.5564335584640503,

0.550839364528656,

0.5456811189651489,

0.5402753353118896,

0.5348427891731262,

0.5297111868858337,

0.5244705080986023,

0.5192984342575073,

0.5144608616828918,

0.5097672343254089,

0.5053302049636841,

0.5007140040397644,

0.4967798590660095,

0.4926583468914032,

0.4885183870792389,

0.4844973683357239,

0.481283038854599,

0.47804296016693115,

0.474494606256485,

0.47126778960227966,

0.4678546190261841,

0.464892715215683,

0.4617881774902344,

0.45883020758628845,

0.45586779713630676,

0.45270147919654846,

0.4497561752796173,

0.4468218684196472,

0.4441770017147064,

0.44138088822364807,

0.4391099214553833,

0.43655523657798767,

0.4336826205253601,

0.43154919147491455,

0.42881661653518677,

0.42638999223709106,

0.42323416471481323,

0.42123183608055115,

0.4187672436237335],

'val\_accuracy': [0.24166665971279144,

0.25,

0.2750000059604645,

0.25833332538604736,

0.3166666626930237,

0.2666666805744171,

0.32499998807907104,

0.34166666865348816,

0.36666667461395264,

0.4416666626930237,

0.46666666865348816,

0.49166667461395264,

0.5,

0.5416666865348816,

0.5333333611488342,

0.5416666865348816,

0.550000011920929,

0.5416666865348816,

0.550000011920929,

0.5833333134651184,

0.6083333492279053,

0.6083333492279053,

0.6166666746139526,

0.625,

0.625,

0.6499999761581421,

0.6666666865348816,

0.675000011920929,

0.6833333373069763,

0.6833333373069763,

0.6916666626930237,

0.6916666626930237,

0.7083333134651184,

0.7250000238418579,

0.7416666746139526,

0.7416666746139526,

0.7333333492279053,

0.7416666746139526,

0.7583333253860474,

0.7666666507720947,

0.7749999761581421,

0.7666666507720947,

0.7833333611488342,

0.7833333611488342,

0.7916666865348816,

0.7916666865348816,

0.7916666865348816,

0.7916666865348816,

0.800000011920929,

0.800000011920929,

0.800000011920929,

0.800000011920929,

0.800000011920929,

0.800000011920929,

0.800000011920929,

0.8083333373069763,

0.8166666626930237,

0.8166666626930237,

0.8166666626930237,

0.8166666626930237,

0.8166666626930237,

0.8166666626930237,

0.8166666626930237,

0.8333333134651184,

0.8333333134651184,

0.8333333134651184,

0.8416666388511658,

0.8416666388511658,

0.8416666388511658,

0.8416666388511658,

0.8416666388511658,

0.8416666388511658,

0.8333333134651184,

0.8333333134651184,

0.8333333134651184,

0.8333333134651184,

0.8333333134651184,

0.8333333134651184,

0.8333333134651184,

0.8333333134651184,

0.8333333134651184,

0.8333333134651184,

0.8416666388511658,

0.8416666388511658,

0.8416666388511658,

0.8500000238418579,

0.8500000238418579,

0.8500000238418579,

0.8500000238418579,

0.8500000238418579,

0.8500000238418579,

0.8416666388511658,

0.8416666388511658,

0.8416666388511658,

0.8416666388511658,

0.8416666388511658,

0.8416666388511658,

0.8500000238418579,

0.8500000238418579,

0.8500000238418579]}

Now visualize the loss over time using history.history:

In [35]:

*# The history.history["loss"] entry is a dictionary with as many values as epochs that the*

*# model was trained on.*

df\_loss\_acc **=** pd.DataFrame(history.history)

df\_loss**=** df\_loss\_acc[['loss','val\_loss']]

df\_loss.rename(columns**=**{'loss':'train','val\_loss':'validation'},inplace**=True**)

df\_acc**=** df\_loss\_acc[['accuracy','val\_accuracy']]

df\_acc.rename(columns**=**{'accuracy':'train','val\_accuracy':'validation'},inplace**=True**)

df\_loss.plot(title**=**'Model loss',figsize**=**(12,8)).set(xlabel**=**'Epoch',ylabel**=**'Loss')

df\_acc.plot(title**=**'Model Accuracy',figsize**=**(12,8)).set(xlabel**=**'Epoch',ylabel**=**'Accuracy')

Out[35]:

[Text(0, 0.5, 'Accuracy'), Text(0.5, 0, 'Epoch')]

Shape

Description automatically generated

Chart, line chart

Description automatically generated

**Congratulations**! You've finished the assignment and built two models: One that recognizes smiles, and another that recognizes SIGN language with almost 80% accuracy on the test set. In addition to that, you now also understand the applications of two Keras APIs: Sequential and Functional. Nicely done!

By now, you know a bit about how the Functional API works and may have glimpsed the possibilities. In your next assignment, you'll really get a feel for its power when you get the opportunity to build a very deep ConvNet, using ResNets!

**6 - Bibliography**

You're always encouraged to read the official documentation. To that end, you can find the docs for the Sequential and Functional APIs here:

<https://www.tensorflow.org/guide/keras/sequential_model>

<https://www.tensorflow.org/guide/keras/functional>