Convolution_model_Application

May 17, 2021

1 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**").

1.1 Table of Contents

1 - Packages

As usual, begin by loading in the packages.

[56]:
```python
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

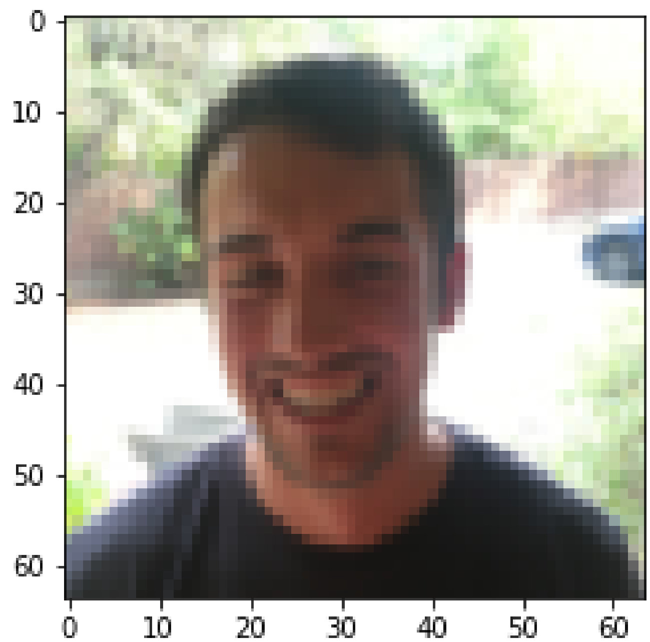
You'll be using the Happy House dataset for this part of the assignment, which contains images of peoples' faces. Your task will be to build a ConvNet that determines whether the people in the images are smiling or not – because they only get to enter the house if they're smiling!

[57]:
```python
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).

[58]:
```
index = 12
plt.imshow(X_train_orig[index]) #display sample training image
plt.show()
```



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

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

- ZeroPadding2D: padding 3, input shape 64 x 64 x 3
- Conv2D: Use 32 7x7 filters, stride 1
- BatchNormalization: for axis 3
- ReLU
- MaxPool2D: Using default parameters
- Flatten the previous output.
- Fully-connected (Dense) layer: Apply a fully connected layer with 1 neuron and a sigmoid activation.

Hint:

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

[67]:
```python
# 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)
    """
    input_shape = (64,64,3)
    model = tf.keras.Sequential([
        tf.keras.Input(shape=input_shape),
            tfl.ZeroPadding2D(
                padding=(3, 3)
            ),
            tfl.Conv2D(
                filters = 32, strides=(1, 1), kernel_size = (7, 7)
            ),
            tfl.BatchNormalization(
                axis= 3
            ),
            tfl.ReLU(),
            tfl.MaxPool2D(
                pool_size=(2, 2), strides=None, padding='valid',␣
↪data_format=None
            ),
            tfl.Flatten(),
            tfl.Dense(
                units = 1, activation='sigmoid'
            )

    ]
    )

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

            #ZeroPadding2D(padding=(3, 3),input_shape = (64,64,3)),
            ## Conv2D with 32 7x7 filters and stride of 1
            #Conv2D(32, 7, strides=1),
            ## BatchNormalization for axis 3
            #BatchNormalization(axis=3),
            ## ReLU
            #Conv2D(activation="relu"),
            ## Max Pooling 2D with default parameters
            #MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',␣
↪data_format=None),
            ## Flatten layer
            #Flatten(),
            #Dense(1,activation='sigmoid')
            ## Dense layer with 1 unit for output & 'sigmoid' activation
```

```
        # YOUR CODE STARTS HERE


    #ZeroPadding2D(padding=(3, 3), data_format=None),
    #Conv2D(32,(7,7),strides=(1,1),input_shape=(64,64,3)),
    #BatchNormalization(axis=3),
    #Activation('relu'),
    #layers.ReLU(max_value=None, negative_slope=0, threshold=0)
    #MaxPooling2D(),
    #Flatten(data_format=None, **kwargs),
    #Dense(1, activation='sigmoid')


        # YOUR CODE ENDS HERE



    return model
```

[68]:

```
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.

[69]:
```
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.

[70]:
```
happy_model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
zero_padding2d_2 (ZeroPaddin (None, 70, 70, 3)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 64, 64, 32)        4736
_________________________________________________________________
batch_normalization_2 (Batch (None, 64, 64, 32)        128
_________________________________________________________________
re_lu_2 (ReLU)               (None, 64, 64, 32)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 32, 32, 32)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 32768)             0
_________________________________________________________________
dense_2 (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).

[71]:
```
happy_model.fit(X_train, Y_train, epochs=10, batch_size=16)
```

```
Epoch 1/10
38/38 [==============================] - 4s 102ms/step - loss: 0.9857 -
accuracy: 0.7033
Epoch 2/10
```

```
38/38 [==============================] - 4s 95ms/step - loss: 0.3051 - accuracy:
0.9000
Epoch 3/10
38/38 [==============================] - 4s 95ms/step - loss: 0.1582 - accuracy:
0.9333
Epoch 4/10
38/38 [==============================] - 4s 97ms/step - loss: 0.1170 - accuracy:
0.9567
Epoch 5/10
38/38 [==============================] - 4s 95ms/step - loss: 0.0876 - accuracy:
0.9717
Epoch 6/10
38/38 [==============================] - 4s 95ms/step - loss: 0.0634 - accuracy:
0.9800
Epoch 7/10
38/38 [==============================] - 4s 95ms/step - loss: 0.0544 - accuracy:
0.9833
Epoch 8/10
38/38 [==============================] - 4s 97ms/step - loss: 0.0833 - accuracy:
0.9700
Epoch 9/10
38/38 [==============================] - 4s 97ms/step - loss: 0.1590 - accuracy:
0.9450
Epoch 10/10
38/38 [==============================] - 4s 95ms/step - loss: 0.1170 - accuracy:
0.9583
```

[71]:
```
<tensorflow.python.keras.callbacks.History at 0x7f575a240250>
```

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.

[72]:
```
happy_model.evaluate(X_test, Y_test)
```

```
5/5 [==============================] - 0s 27ms/step - loss: 0.2695 - accuracy:
0.8667
```

[72]:
```
[0.26954564452171326, 0.8666666746139526]
```

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 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

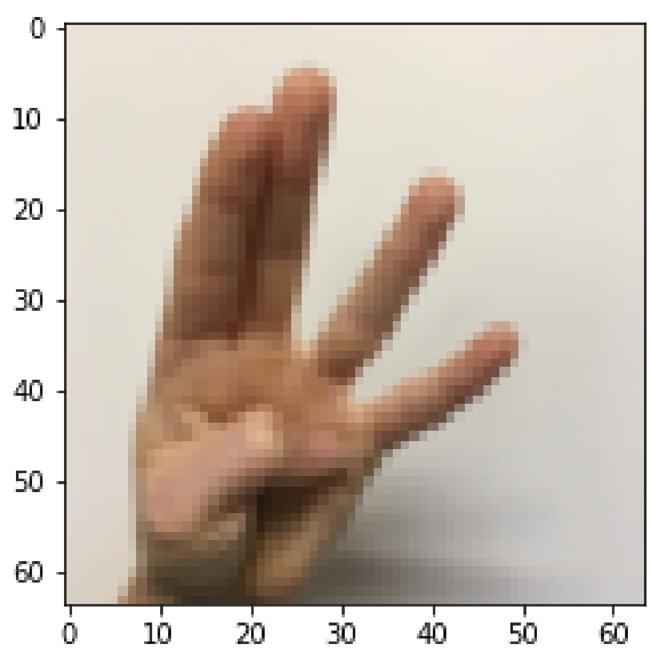
As a reminder, the SIGNS dataset is a collection of 6 signs representing numbers from 0 to 5.

[73]:
```
# 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.

[74]:
```
# 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
```



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.

[75]:
```
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, 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.
- **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.
- **tf.keras.layers.ReLU():** computes the elementwise ReLU of Z (which can be any shape). You can read the full documentation on 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 $k = h \times w \times c$. "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.
- **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.

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: Use 8 4 by 4 filters, stride 1, padding is "SAME"
- ReLU
- 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 the previous output.
- Fully-connected (Dense) layer: Apply a fully connected layer with 6 neurons and a softmax activation.

[76]:
```
# GRADED FUNCTION: convolutional_model
```

```python
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 = None
    ## RELU
    # A1 = None
    ## MAXPOOL: window 8x8, stride 8, padding 'SAME'
    # P1 = None
    ## CONV2D: 16 filters 2x2, stride 1, padding 'SAME'
    # Z2 = None
    ## RELU
    # A2 = None
    ## MAXPOOL: window 4x4, stride 4, padding 'SAME'
    # P2 = None
    ## FLATTEN
    # F = None
    ## Dense layer
    ## 6 neurons in output layer. Hint: one of the arguments should be␣
 ↪"activation='softmax'"
    # outputs = None
    # YOUR CODE STARTS HERE
    Z1 = tfl.Conv2D(8, (4, 4), strides=(1, 1), padding="same")(input_img)
    A1 = tfl.ReLU()(Z1)
    P1 = tfl.MaxPooling2D(pool_size=(8, 8), strides=(8, 8), padding='same')(A1)
    Z2 = tfl.Conv2D(16, (2,2), strides=(1, 1), padding="same")(P1)
    A2 = tfl.ReLU()(Z2)
    P2 = tfl.MaxPooling2D(pool_size=(4, 4), strides=(4, 4), padding='same')(A2)
    F = tfl.Flatten()(P2)
    outputs = tfl.Dense(units = 6, activation='softmax')(F)

    # YOUR CODE ENDS HERE
```

```python
    model = tf.keras.Model(inputs=input_img, outputs=outputs)
    return model
```

[77]:

```python
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_5 (InputLayer)         [(None, 64, 64, 3)]       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 64, 64, 8)         392
_________________________________________________________________
re_lu_3 (ReLU)               (None, 64, 64, 8)         0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 16)          528
_________________________________________________________________
re_lu_4 (ReLU)               (None, 8, 8, 16)          0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 2, 2, 16)          0
_________________________________________________________________
flatten_3 (Flatten)          (None, 64)                0
_________________________________________________________________
dense_3 (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

[78]:
```
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 112ms/step - loss: 1.8240 -
accuracy: 0.1806 - val_loss: 1.8071 - val_accuracy: 0.2000
Epoch 2/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7922 -
accuracy: 0.2028 - val_loss: 1.7920 - val_accuracy: 0.1083
Epoch 3/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7843 -
accuracy: 0.1648 - val_loss: 1.7873 - val_accuracy: 0.1500
Epoch 4/100
17/17 [==============================] - 2s 111ms/step - loss: 1.7766 -
accuracy: 0.2565 - val_loss: 1.7848 - val_accuracy: 0.2083
Epoch 5/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7692 -
accuracy: 0.2676 - val_loss: 1.7793 - val_accuracy: 0.2167
Epoch 6/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7592 -
accuracy: 0.3000 - val_loss: 1.7710 - val_accuracy: 0.2417
Epoch 7/100
17/17 [==============================] - 2s 107ms/step - loss: 1.7476 -
accuracy: 0.3296 - val_loss: 1.7619 - val_accuracy: 0.2750
Epoch 8/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7335 -
accuracy: 0.3620 - val_loss: 1.7505 - val_accuracy: 0.2833
Epoch 9/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7167 -
accuracy: 0.3833 - val_loss: 1.7358 - val_accuracy: 0.3167
Epoch 10/100
17/17 [==============================] - 2s 106ms/step - loss: 1.6968 -
accuracy: 0.4009 - val_loss: 1.7193 - val_accuracy: 0.3500
Epoch 11/100
17/17 [==============================] - 2s 111ms/step - loss: 1.6736 -
accuracy: 0.4056 - val_loss: 1.7002 - val_accuracy: 0.3500
Epoch 12/100
17/17 [==============================] - 2s 107ms/step - loss: 1.6460 -
accuracy: 0.4463 - val_loss: 1.6777 - val_accuracy: 0.4083
Epoch 13/100
```

```
17/17 [==============================] - 2s 106ms/step - loss: 1.6161 -
accuracy: 0.4463 - val_loss: 1.6502 - val_accuracy: 0.4167
Epoch 14/100
17/17 [==============================] - 2s 111ms/step - loss: 1.5819 -
accuracy: 0.4759 - val_loss: 1.6168 - val_accuracy: 0.4417
Epoch 15/100
17/17 [==============================] - 2s 111ms/step - loss: 1.5437 -
accuracy: 0.4852 - val_loss: 1.5813 - val_accuracy: 0.4667
Epoch 16/100
17/17 [==============================] - 2s 111ms/step - loss: 1.5045 -
accuracy: 0.4917 - val_loss: 1.5471 - val_accuracy: 0.4833
Epoch 17/100
17/17 [==============================] - 2s 106ms/step - loss: 1.4655 -
accuracy: 0.4926 - val_loss: 1.5115 - val_accuracy: 0.4833
Epoch 18/100
17/17 [==============================] - 2s 106ms/step - loss: 1.4262 -
accuracy: 0.5167 - val_loss: 1.4758 - val_accuracy: 0.4833
Epoch 19/100
17/17 [==============================] - 2s 106ms/step - loss: 1.3864 -
accuracy: 0.5306 - val_loss: 1.4400 - val_accuracy: 0.4750
Epoch 20/100
17/17 [==============================] - 2s 111ms/step - loss: 1.3467 -
accuracy: 0.5444 - val_loss: 1.4007 - val_accuracy: 0.5083
Epoch 21/100
17/17 [==============================] - 2s 106ms/step - loss: 1.3070 -
accuracy: 0.5620 - val_loss: 1.3611 - val_accuracy: 0.5417
Epoch 22/100
17/17 [==============================] - 2s 106ms/step - loss: 1.2673 -
accuracy: 0.5759 - val_loss: 1.3226 - val_accuracy: 0.5667
Epoch 23/100
17/17 [==============================] - 2s 112ms/step - loss: 1.2295 -
accuracy: 0.5861 - val_loss: 1.2845 - val_accuracy: 0.5750
Epoch 24/100
17/17 [==============================] - 2s 112ms/step - loss: 1.1928 -
accuracy: 0.6065 - val_loss: 1.2451 - val_accuracy: 0.5833
Epoch 25/100
17/17 [==============================] - 2s 111ms/step - loss: 1.1572 -
accuracy: 0.6157 - val_loss: 1.2073 - val_accuracy: 0.6000
Epoch 26/100
17/17 [==============================] - 2s 111ms/step - loss: 1.1238 -
accuracy: 0.6287 - val_loss: 1.1729 - val_accuracy: 0.6167
Epoch 27/100
17/17 [==============================] - 2s 112ms/step - loss: 1.0924 -
accuracy: 0.6398 - val_loss: 1.1412 - val_accuracy: 0.6250
Epoch 28/100
17/17 [==============================] - 2s 106ms/step - loss: 1.0625 -
accuracy: 0.6556 - val_loss: 1.1115 - val_accuracy: 0.6333
Epoch 29/100
```

```
17/17 [==============================] - 2s 107ms/step - loss: 1.0342 -
accuracy: 0.6685 - val_loss: 1.0825 - val_accuracy: 0.6333
Epoch 30/100
17/17 [==============================] - 2s 106ms/step - loss: 1.0070 -
accuracy: 0.6741 - val_loss: 1.0549 - val_accuracy: 0.6333
Epoch 31/100
17/17 [==============================] - 2s 107ms/step - loss: 0.9817 -
accuracy: 0.6870 - val_loss: 1.0317 - val_accuracy: 0.6250
Epoch 32/100
17/17 [==============================] - 2s 106ms/step - loss: 0.9576 -
accuracy: 0.6963 - val_loss: 1.0093 - val_accuracy: 0.6333
Epoch 33/100
17/17 [==============================] - 2s 111ms/step - loss: 0.9343 -
accuracy: 0.7065 - val_loss: 0.9853 - val_accuracy: 0.6500
Epoch 34/100
17/17 [==============================] - 2s 111ms/step - loss: 0.9117 -
accuracy: 0.7213 - val_loss: 0.9646 - val_accuracy: 0.6667
Epoch 35/100
17/17 [==============================] - 2s 111ms/step - loss: 0.8904 -
accuracy: 0.7296 - val_loss: 0.9444 - val_accuracy: 0.6667
Epoch 36/100
17/17 [==============================] - 2s 111ms/step - loss: 0.8697 -
accuracy: 0.7380 - val_loss: 0.9262 - val_accuracy: 0.6833
Epoch 37/100
17/17 [==============================] - 2s 106ms/step - loss: 0.8505 -
accuracy: 0.7435 - val_loss: 0.9078 - val_accuracy: 0.6917
Epoch 38/100
17/17 [==============================] - 2s 111ms/step - loss: 0.8320 -
accuracy: 0.7491 - val_loss: 0.8900 - val_accuracy: 0.6917
Epoch 39/100
17/17 [==============================] - 2s 106ms/step - loss: 0.8151 -
accuracy: 0.7556 - val_loss: 0.8739 - val_accuracy: 0.6833
Epoch 40/100
17/17 [==============================] - 2s 106ms/step - loss: 0.7982 -
accuracy: 0.7639 - val_loss: 0.8574 - val_accuracy: 0.7083
Epoch 41/100
17/17 [==============================] - 2s 111ms/step - loss: 0.7817 -
accuracy: 0.7685 - val_loss: 0.8407 - val_accuracy: 0.7167
Epoch 42/100
17/17 [==============================] - 2s 106ms/step - loss: 0.7658 -
accuracy: 0.7713 - val_loss: 0.8242 - val_accuracy: 0.7167
Epoch 43/100
17/17 [==============================] - 2s 106ms/step - loss: 0.7509 -
accuracy: 0.7806 - val_loss: 0.8106 - val_accuracy: 0.7167
Epoch 44/100
17/17 [==============================] - 2s 106ms/step - loss: 0.7366 -
accuracy: 0.7852 - val_loss: 0.7981 - val_accuracy: 0.7250
Epoch 45/100
```

```
17/17 [==============================] - 2s 106ms/step - loss: 0.7232 -
accuracy: 0.7861 - val_loss: 0.7848 - val_accuracy: 0.7417
Epoch 46/100
17/17 [==============================] - 2s 107ms/step - loss: 0.7101 -
accuracy: 0.7889 - val_loss: 0.7730 - val_accuracy: 0.7500
Epoch 47/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6979 -
accuracy: 0.7926 - val_loss: 0.7618 - val_accuracy: 0.7583
Epoch 48/100
17/17 [==============================] - 2s 111ms/step - loss: 0.6859 -
accuracy: 0.7944 - val_loss: 0.7519 - val_accuracy: 0.7583
Epoch 49/100
17/17 [==============================] - 2s 111ms/step - loss: 0.6744 -
accuracy: 0.7972 - val_loss: 0.7419 - val_accuracy: 0.7583
Epoch 50/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6639 -
accuracy: 0.8009 - val_loss: 0.7319 - val_accuracy: 0.7417
Epoch 51/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6535 -
accuracy: 0.8037 - val_loss: 0.7226 - val_accuracy: 0.7417
Epoch 52/100
17/17 [==============================] - 2s 112ms/step - loss: 0.6435 -
accuracy: 0.8046 - val_loss: 0.7134 - val_accuracy: 0.7500
Epoch 53/100
17/17 [==============================] - 2s 111ms/step - loss: 0.6343 -
accuracy: 0.8093 - val_loss: 0.7050 - val_accuracy: 0.7500
Epoch 54/100
17/17 [==============================] - 2s 107ms/step - loss: 0.6247 -
accuracy: 0.8120 - val_loss: 0.6975 - val_accuracy: 0.7500
Epoch 55/100
17/17 [==============================] - 2s 107ms/step - loss: 0.6160 -
accuracy: 0.8176 - val_loss: 0.6898 - val_accuracy: 0.7500
Epoch 56/100
17/17 [==============================] - 2s 112ms/step - loss: 0.6071 -
accuracy: 0.8176 - val_loss: 0.6830 - val_accuracy: 0.7500
Epoch 57/100
17/17 [==============================] - 2s 107ms/step - loss: 0.5988 -
accuracy: 0.8194 - val_loss: 0.6765 - val_accuracy: 0.7500
Epoch 58/100
17/17 [==============================] - 2s 112ms/step - loss: 0.5910 -
accuracy: 0.8185 - val_loss: 0.6701 - val_accuracy: 0.7417
Epoch 59/100
17/17 [==============================] - 2s 111ms/step - loss: 0.5836 -
accuracy: 0.8194 - val_loss: 0.6644 - val_accuracy: 0.7417
Epoch 60/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5762 -
accuracy: 0.8222 - val_loss: 0.6581 - val_accuracy: 0.7500
Epoch 61/100
```

```
17/17 [==============================] - 2s 106ms/step - loss: 0.5691 -
accuracy: 0.8231 - val_loss: 0.6541 - val_accuracy: 0.7417
Epoch 62/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5627 -
accuracy: 0.8250 - val_loss: 0.6492 - val_accuracy: 0.7417
Epoch 63/100
17/17 [==============================] - 2s 111ms/step - loss: 0.5562 -
accuracy: 0.8269 - val_loss: 0.6460 - val_accuracy: 0.7417
Epoch 64/100
17/17 [==============================] - 2s 112ms/step - loss: 0.5501 -
accuracy: 0.8278 - val_loss: 0.6404 - val_accuracy: 0.7417
Epoch 65/100
17/17 [==============================] - 2s 107ms/step - loss: 0.5437 -
accuracy: 0.8306 - val_loss: 0.6362 - val_accuracy: 0.7583
Epoch 66/100
17/17 [==============================] - 2s 107ms/step - loss: 0.5374 -
accuracy: 0.8315 - val_loss: 0.6316 - val_accuracy: 0.7583
Epoch 67/100
17/17 [==============================] - 2s 112ms/step - loss: 0.5313 -
accuracy: 0.8333 - val_loss: 0.6289 - val_accuracy: 0.7667
Epoch 68/100
17/17 [==============================] - 2s 111ms/step - loss: 0.5255 -
accuracy: 0.8333 - val_loss: 0.6238 - val_accuracy: 0.7750
Epoch 69/100
17/17 [==============================] - 2s 107ms/step - loss: 0.5198 -
accuracy: 0.8352 - val_loss: 0.6207 - val_accuracy: 0.7750
Epoch 70/100
17/17 [==============================] - 2s 107ms/step - loss: 0.5143 -
accuracy: 0.8370 - val_loss: 0.6165 - val_accuracy: 0.7833
Epoch 71/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5091 -
accuracy: 0.8407 - val_loss: 0.6121 - val_accuracy: 0.7833
Epoch 72/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5034 -
accuracy: 0.8435 - val_loss: 0.6087 - val_accuracy: 0.7833
Epoch 73/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4983 -
accuracy: 0.8454 - val_loss: 0.6048 - val_accuracy: 0.7833
Epoch 74/100
17/17 [==============================] - 2s 111ms/step - loss: 0.4933 -
accuracy: 0.8454 - val_loss: 0.6010 - val_accuracy: 0.7833
Epoch 75/100
17/17 [==============================] - 2s 107ms/step - loss: 0.4882 -
accuracy: 0.8472 - val_loss: 0.5963 - val_accuracy: 0.7833
Epoch 76/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4828 -
accuracy: 0.8500 - val_loss: 0.5913 - val_accuracy: 0.7833
Epoch 77/100
```

```
17/17 [==============================] - 2s 106ms/step - loss: 0.4777 -
accuracy: 0.8556 - val_loss: 0.5870 - val_accuracy: 0.7833
Epoch 78/100
17/17 [==============================] - 2s 111ms/step - loss: 0.4732 -
accuracy: 0.8546 - val_loss: 0.5825 - val_accuracy: 0.7833
Epoch 79/100
17/17 [==============================] - 2s 112ms/step - loss: 0.4685 -
accuracy: 0.8565 - val_loss: 0.5796 - val_accuracy: 0.7750
Epoch 80/100
17/17 [==============================] - 2s 112ms/step - loss: 0.4637 -
accuracy: 0.8593 - val_loss: 0.5770 - val_accuracy: 0.7667
Epoch 81/100
17/17 [==============================] - 2s 107ms/step - loss: 0.4592 -
accuracy: 0.8593 - val_loss: 0.5734 - val_accuracy: 0.7667
Epoch 82/100
17/17 [==============================] - 2s 107ms/step - loss: 0.4547 -
accuracy: 0.8602 - val_loss: 0.5716 - val_accuracy: 0.7583
Epoch 83/100
17/17 [==============================] - 2s 107ms/step - loss: 0.4504 -
accuracy: 0.8620 - val_loss: 0.5679 - val_accuracy: 0.7583
Epoch 84/100
17/17 [==============================] - 2s 107ms/step - loss: 0.4462 -
accuracy: 0.8639 - val_loss: 0.5648 - val_accuracy: 0.7583
Epoch 85/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4421 -
accuracy: 0.8657 - val_loss: 0.5620 - val_accuracy: 0.7667
Epoch 86/100
17/17 [==============================] - 2s 112ms/step - loss: 0.4379 -
accuracy: 0.8657 - val_loss: 0.5596 - val_accuracy: 0.7750
Epoch 87/100
17/17 [==============================] - 2s 112ms/step - loss: 0.4338 -
accuracy: 0.8694 - val_loss: 0.5579 - val_accuracy: 0.7667
Epoch 88/100
17/17 [==============================] - 2s 111ms/step - loss: 0.4297 -
accuracy: 0.8694 - val_loss: 0.5545 - val_accuracy: 0.7750
Epoch 89/100
17/17 [==============================] - 2s 111ms/step - loss: 0.4258 -
accuracy: 0.8713 - val_loss: 0.5529 - val_accuracy: 0.7750
Epoch 90/100
17/17 [==============================] - 2s 112ms/step - loss: 0.4219 -
accuracy: 0.8713 - val_loss: 0.5505 - val_accuracy: 0.7917
Epoch 91/100
17/17 [==============================] - 2s 107ms/step - loss: 0.4181 -
accuracy: 0.8731 - val_loss: 0.5484 - val_accuracy: 0.7917
Epoch 92/100
17/17 [==============================] - 2s 112ms/step - loss: 0.4143 -
accuracy: 0.8750 - val_loss: 0.5461 - val_accuracy: 0.7917
Epoch 93/100
```

```
17/17 [==============================] - 2s 106ms/step - loss: 0.4106 -
accuracy: 0.8759 - val_loss: 0.5434 - val_accuracy: 0.8000
Epoch 94/100
17/17 [==============================] - 2s 112ms/step - loss: 0.4068 -
accuracy: 0.8750 - val_loss: 0.5411 - val_accuracy: 0.8000
Epoch 95/100
17/17 [==============================] - 2s 112ms/step - loss: 0.4034 -
accuracy: 0.8787 - val_loss: 0.5386 - val_accuracy: 0.8000
Epoch 96/100
17/17 [==============================] - 2s 112ms/step - loss: 0.3997 -
accuracy: 0.8815 - val_loss: 0.5365 - val_accuracy: 0.8000
Epoch 97/100
17/17 [==============================] - 2s 111ms/step - loss: 0.3962 -
accuracy: 0.8815 - val_loss: 0.5349 - val_accuracy: 0.8167
Epoch 98/100
17/17 [==============================] - 2s 112ms/step - loss: 0.3927 -
accuracy: 0.8833 - val_loss: 0.5321 - val_accuracy: 0.8250
Epoch 99/100
17/17 [==============================] - 2s 112ms/step - loss: 0.3893 -
accuracy: 0.8833 - val_loss: 0.5304 - val_accuracy: 0.8167
Epoch 100/100
17/17 [==============================] - 2s 106ms/step - loss: 0.3859 -
accuracy: 0.8880 - val_loss: 0.5283 - val_accuracy: 0.8167
```

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`:

[79]:
```
history.history
```

[79]:
```
{'loss': [1.8239885568618774,
  1.792216420173645,
  1.7842594385147095,
  1.7766059637069702,
  1.769153356552124,
  1.759247064590454,
  1.747624397277832,
  1.7335017919540405,
  1.7166863679885864,
  1.6968001127243042,
  1.6736479997634888,
  1.6460130214691162,
  1.6160879135131836,
  1.5818753242492676,
  1.5437226295471191,
  1.504463791847229,
  1.4655016660690308,
```

```
1.4261634349822998,
1.3864285945892334,
1.346679210662841 8,
1.3069730997085571,
1.2673072814941406,
1.2295434474945068,
1.1927632093429565,
1.1572076082229614,
1.1237709522247314,
1.092442274093628,
1.062488317489624,
1.0342355966567993,
1.0070399045944214,
0.9816875457763672,
0.9575812220573425,
0.9342584013938904,
0.9116811156272888,
0.8903510570526123,
0.8697140216827393,
0.8504648208618164,
0.8320045471191406,
0.8150596022605896,
0.7982215285301208,
0.7817295789718628,
0.765774667263031,
0.750862181186676,
0.7365915179252625,
0.7231794595718384,
0.7100703716278076,
0.6979115605354309,
0.6859374046325684,
0.674444317817688,
0.6639037132263184,
0.6534873843193054,
0.6434926390647888,
0.6343183517456055,
0.6247369050979614,
0.6159927845001221,
0.6071168184280396,
0.5988014340400696,
0.5910252928733826,
0.5836165547370911,
0.5762375593185425,
0.5691040754318237,
0.5626841187477112,
0.5561703443527222,
0.5500537157058716,
```

```
 0.5436879396438599,
 0.5373609066009521,
 0.531264066696167,
 0.5254679918289185,
 0.5198062062263489,
 0.5143393278121948,
 0.5090556740760803,
 0.5034219026565552,
 0.4983309209346771,
 0.4933498203754425,
 0.4882037043571472,
 0.4828338623046875,
 0.4777493476867676,
 0.47318580746650696,
 0.4684942066669464,
 0.4637395143508911,
 0.4591827392578125,
 0.4547412693500519,
 0.4503907263278961,
 0.44624489545822144,
 0.4420773386955261,
 0.4378516376018524,
 0.43380698561668396,
 0.4297111928462982,
 0.4258488118648529,
 0.4218570291996002,
 0.41812074184417725,
 0.4143238961696625,
 0.41064372658729553,
 0.40684109926223755,
 0.4033716917037964,
 0.39965084195137024,
 0.3962344229221344,
 0.3927242159843445,
 0.3893465995788574,
 0.3858858048915863],
'accuracy': [0.1805555522441864,
 0.20277777314186096,
 0.1648148149251938,
 0.25648146867752075,
 0.26759257912635803,
 0.30000001192092896,
 0.3296296298503876,
 0.3620370328426361,
 0.38333332538604736,
 0.4009259343147278,
 0.4055555462837219,
```

0.4462963044643402,
0.4462963044643402,
0.47592592239379883,
0.4851851761341095,
0.49166667461395264,
0.49259260296821594,
0.5166666507720947,
0.5305555462837219,
0.5444444417953491,
0.5620370507240295,
0.5759259462356567,
0.5861111283302307,
0.6064814925193787,
0.6157407164573669,
0.6287037134170532,
0.6398147940635681,
0.6555555462837219,
0.6685185432434082,
0.6740740537643433,
0.6870370507240295,
0.6962962746620178,
0.7064814567565918,
0.7212963104248047,
0.729629635810852,
0.7379629611968994,
0.7435185313224792,
0.7490741014480591,
0.7555555701255798,
0.7638888955116272,
0.7685185074806213,
0.7712963223457336,
0.7805555462837219,
0.7851851582527161,
0.7861111164093018,
0.7888888716697693,
0.7925925850868225,
0.7944444417953491,
0.7972221970558167,
0.8009259104728699,
0.8037037253379822,
0.8046296238899231,
0.8092592358589172,
0.8120370507240295,
0.8175926208496094,
0.8175926208496094,
0.8194444179534912,
0.8185185194015503,

```
 0.8194444179534912,
 0.8222222328186035,
 0.8231481313705444,
 0.824999988079071,
 0.8268518447875977,
 0.8277778029441833,
 0.8305555582046509,
 0.8314814567565918,
 0.8333333134651184,
 0.8333333134651184,
 0.835185170173645,
 0.8370370268821716,
 0.8407407402992249,
 0.8435184955596924,
 0.845370352268219,
 0.845370352268219,
 0.8472222089767456,
 0.8500000238418579,
 0.855555534362793,
 0.854629635810852,
 0.8564814925193787,
 0.8592592477798462,
 0.8592592477798462,
 0.8601852059364319,
 0.8620370626449585,
 0.8638888597488403,
 0.8657407164573669,
 0.8657407164573669,
 0.8694444298744202,
 0.8694444298744202,
 0.8712962865829468,
 0.8712962865829468,
 0.8731481432914734,
 0.875,
 0.8759258985519409,
 0.875,
 0.8787037134170532,
 0.8814814686775208,
 0.8814814686775208,
 0.8833333253860474,
 0.8833333253860474,
 0.8879629373550415],
'val_loss': [1.8071242570877075,
 1.7920087575912476,
 1.7872673273086548,
 1.7848368883132935,
 1.7793126106262207,
```

```
1.771008849143982,
1.761905550956726,
1.7504719495773315,
1.7357561588287354,
1.7193472385406494,
1.7002135515213013,
1.6776849031448364,
1.6501673460006714,
1.6168311834335327,
1.5813477039337158,
1.547104120254516б,
1.5115257501602173,
1.4757750034332275,
1.4400486946105957,
1.4006654024124146,
1.3611301183700562,
1.3225748538970947,
1.2844970226287842,
1.2450871467590332,
1.2072632312774658,
1.1729215383529663,
1.1411681175231934,
1.111480474472046,
1.0824769735336304,
1.0548871755599976,
1.0316623449325562,
1.009300708770752,
0.9853000640869141,
0.9646235108375549,
0.9444288611412048,
0.9262329339981079,
0.9077568054199219,
0.8900398015975952,
0.8739340305328369,
0.8573685884475708,
0.8406862020492554,
0.8241629004478455,
0.8105897307395935,
0.7980601191520691,
0.7847877144813538,
0.7730434536933899,
0.7617528438568115,
0.7518805265426636,
0.741934061050415,
0.7319008111953735,
0.722604513168335,
0.7134038209915161,
```

```
0.7049983739852905,
0.6975072026252747,
0.6897673010826111,
0.6830332279205322,
0.6765353083610535,
0.6701333522796631,
0.6644311547279358,
0.6581292748451233,
0.6541017889976501,
0.6492155194282532,
0.6459954380989075,
0.6403607726097107,
0.636218249797821,
0.6316276788711548,
0.6289151310920715,
0.6237677931785583,
0.6207249164581299,
0.6165114045143127,
0.6120990514755249,
0.6087333559989929,
0.6048375368118286,
0.6009560227394104,
0.5963115692138672,
0.5913121700286865,
0.5869784951210022,
0.582494854927063,
0.5795844197273254,
0.5769504308700562,
0.5734025239944458,
0.5716027617454529,
0.5678818821907043,
0.5647956132888794,
0.5619531273841858,
0.5596122145652771,
0.5578969120979309,
0.5545381903648376,
0.5528603792190552,
0.5505332946777344,
0.5483893752098083,
0.5460778474807739,
0.5433598756790161,
0.5411055088043213,
0.538575291633606,
0.5365241765975952,
0.5348969101905823,
0.5321115851402283,
0.5304465889930725,
```

```
 0.5282604694366455],
'val_accuracy': [0.20000000298023224,
 0.10833333432674408,
 0.15000000596046448,
 0.2083333283662796,
 0.21666666865348816,
 0.24166665971279144,
 0.2750000059604645,
 0.28333333134651184,
 0.3166666626930237,
 0.3499999940395355,
 0.3499999940395355,
 0.40833333134651184,
 0.4166666567325592,
 0.4416666626930237,
 0.46666666865348816,
 0.4833333194255829,
 0.4833333194255829,
 0.4833333194255829,
 0.4749999940395355,
 0.5083333253860474,
 0.5416666865348816,
 0.5666666626930237,
 0.574999988079071,
 0.5833333134651184,
 0.6000000238418579,
 0.6166666746139526,
 0.625,
 0.6333333253860474,
 0.6333333253860474,
 0.6333333253860474,
 0.625,
 0.6333333253860474,
 0.6499999761581421,
 0.6666666865348816,
 0.6666666865348816,
 0.6833333373069763,
 0.6916666626930237,
 0.6916666626930237,
 0.6833333373069763,
 0.7083333134651184,
 0.7166666388511658,
 0.7166666388511658,
 0.7166666388511658,
 0.7250000238418579,
 0.7416666746139526,
 0.75,
```

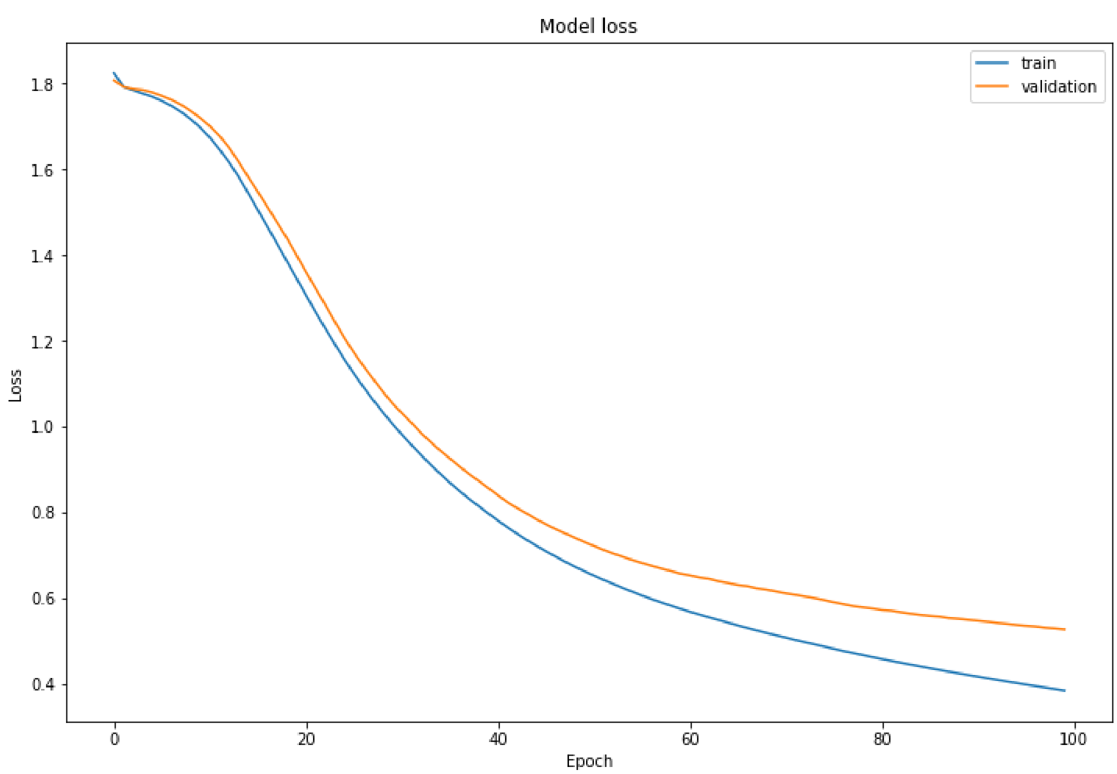
```
0.7583333253860474,
0.7583333253860474,
0.7583333253860474,
0.7416666746139526,
0.7416666746139526,
0.75,
0.75,
0.75,
0.75,
0.75,
0.75,
0.7416666746139526,
0.7416666746139526,
0.75,
0.7416666746139526,
0.7416666746139526,
0.7416666746139526,
0.7416666746139526,
0.7583333253860474,
0.7583333253860474,
0.7666666507720947,
0.7749999761581421,
0.7749999761581421,
0.7833333611488342,
0.7833333611488342,
0.7833333611488342,
0.7833333611488342,
0.7833333611488342,
0.7833333611488342,
0.7833333611488342,
0.7833333611488342,
0.7833333611488342,
0.7749999761581421,
0.7666666507720947,
0.7666666507720947,
0.7583333253860474,
0.7583333253860474,
0.7583333253860474,
0.7666666507720947,
0.7749999761581421,
0.7666666507720947,
0.7749999761581421,
0.7749999761581421,
0.7916666865348816,
0.7916666865348816,
0.7916666865348816,
0.800000011920929,
```

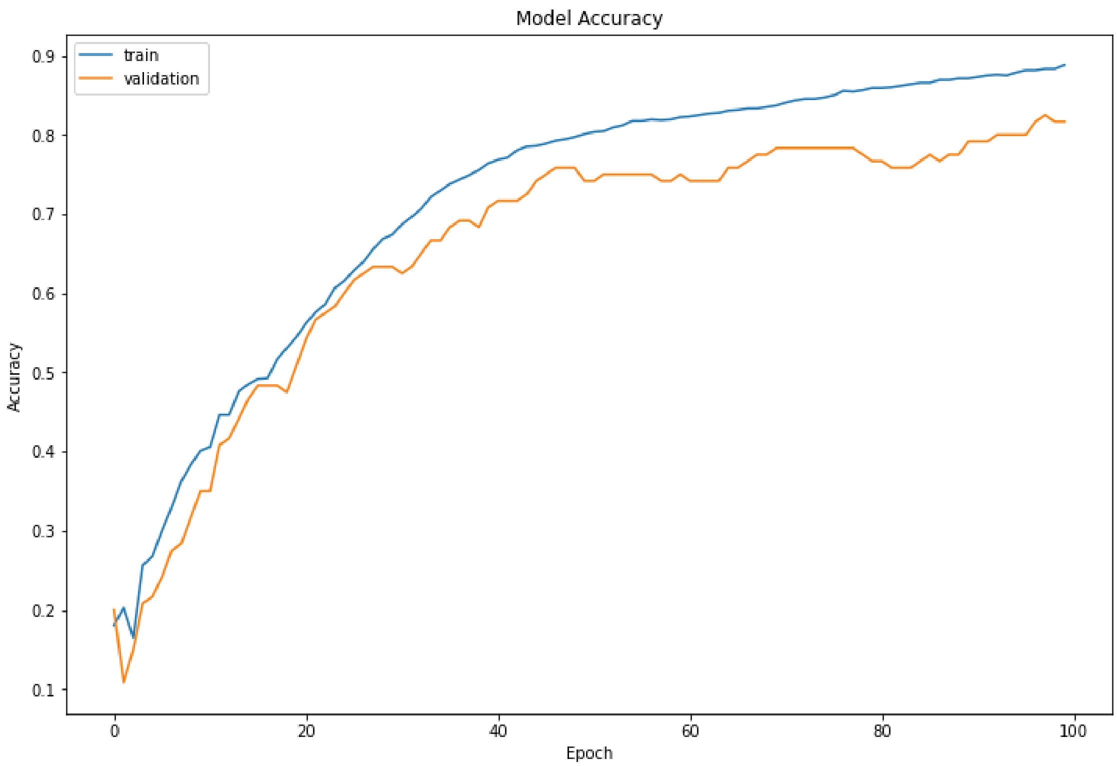
```
0.800000011920929,
0.800000011920929,
0.800000011920929,
0.8166666626930237,
0.824999988079071,
0.8166666626930237,
0.8166666626930237]}
```

Now visualize the loss over time using `history.history`:

[80]:
```
# 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')
```

[80]:
```
[Text(0, 0.5, 'Accuracy'), Text(0.5, 0, 'Epoch')]
```





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