

Convolution_model_Application

February 7, 2025

1 Convolutional Neural Networks: Application

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

- Create a mood classifier 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 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 Error: Grader feedback 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](#).

1.2 Table of Contents

- Section ??
 - Section ??

- Section ??
- Section ??
 - Section ??
 - * Section ??
 - Section ??
- Section ??
 - Section ??
 - Section ??
 - Section ??
 - * Section ??
 - Section ??
- Section ??
- Section ??

1 - Packages

As usual, begin by loading in the packages.

```
[ ]: ### v1.1
```

```
[1]: 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!

```
[2]: 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)

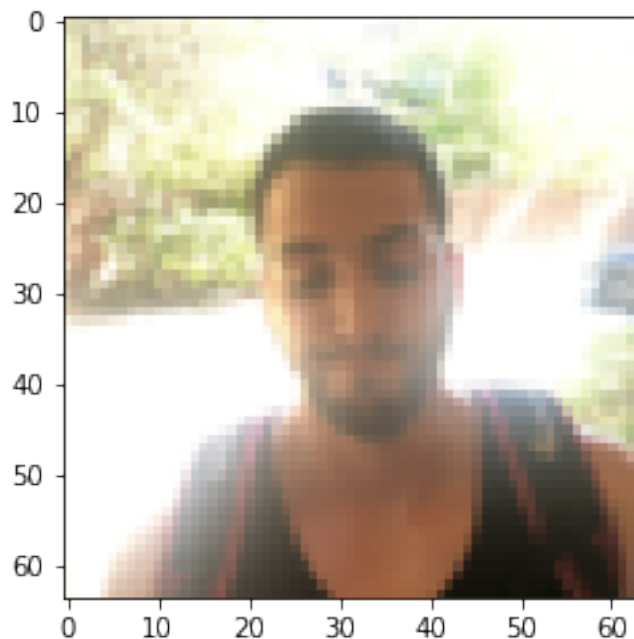
```

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

```

[3]: index = 124
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**

```
[4]: # 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

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

    #     ## BatchNormalization for axis 3

    #     ## ReLU

    #     ## Max Pooling 2D with default parameters

    #     ## Flatten layer

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

    #     # YOUR CODE STARTS HERE

    #     # YOUR CODE ENDS HERE
    # ])
    model = tf.keras.Sequential([
        # ZeroPadding2D layer
```

```

        tf.keras.layers.ZeroPadding2D(padding=(3, 3), input_shape=(64, 64,
→3)),

        # Conv2D layer
        tf.keras.layers.Conv2D(filters=32, kernel_size=(7, 7), strides=(1,
→1), padding='valid'),

        # BatchNormalization layer
        tf.keras.layers.BatchNormalization(axis=3),

        # ReLU activation
        tf.keras.layers.ReLU(),

        # MaxPooling2D layer
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),

        # Flatten layer
        tf.keras.layers.Flatten(),

        # Dense layer with 1 output unit and sigmoid activation
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])
    return model

```

```

[5]: 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!

Expected 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](#).

```
[6]: 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.

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

Model: "sequential"

Layer (type)	Output Shape	Param #
zero_padding2d (ZeroPadding2D)	(None, 70, 70, 3)	0
conv2d (Conv2D)	(None, 64, 64, 32)	4736
batch_normalization (Batch Normalization)	(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).

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

```
Epoch 1/10
38/38 [=====] - 4s 100ms/step - loss: 1.1936 -
accuracy: 0.6600
Epoch 2/10
38/38 [=====] - 4s 97ms/step - loss: 0.2724 - accuracy:
0.9067
Epoch 3/10
38/38 [=====] - 4s 95ms/step - loss: 0.2744 - accuracy:
0.8967
Epoch 4/10
38/38 [=====] - 4s 97ms/step - loss: 0.1979 - accuracy:
0.9200
Epoch 5/10
38/38 [=====] - 4s 98ms/step - loss: 0.1170 - accuracy:
0.9617
Epoch 6/10
38/38 [=====] - 4s 100ms/step - loss: 0.1418 -
accuracy: 0.9533
Epoch 7/10
38/38 [=====] - 4s 95ms/step - loss: 0.0875 - accuracy:
0.9617
Epoch 8/10
38/38 [=====] - 4s 97ms/step - loss: 0.1110 - accuracy:
0.9500
Epoch 9/10
38/38 [=====] - 4s 95ms/step - loss: 0.1003 - accuracy:
0.9517
Epoch 10/10
38/38 [=====] - 4s 95ms/step - loss: 0.1264 - accuracy:
0.9567
```

```
[8]: <tensorflow.python.keras.callbacks.History at 0x7ef36351e510>
```

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.

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

```
5/5 [=====] - 0s 25ms/step - loss: 0.1790 - accuracy: 0.9200
```

```
[9]: [0.1790035516023636, 0.9200000166893005]
```

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

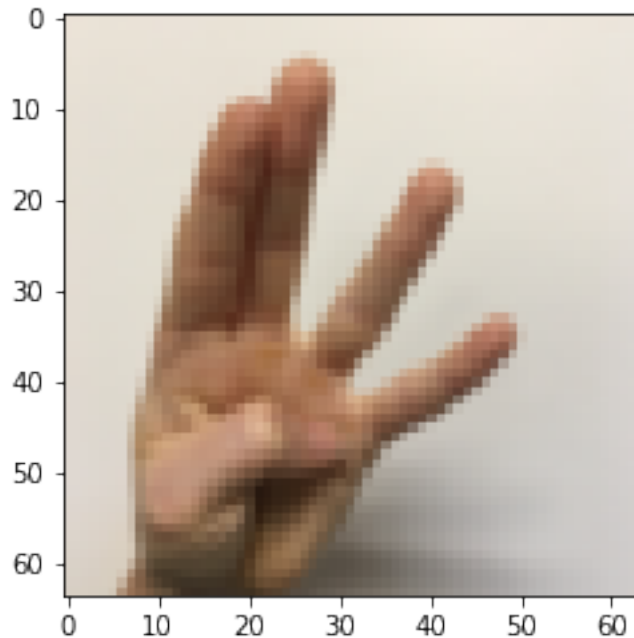
As a reminder, the SIGNS dataset is a collection of 6 signs representing numbers from 0 to 5.

```
[10]: # 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.

```
[11]: # 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.

```
[12]: 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.

```
[15]: # 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 = 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

input_img = tf.keras.Input(shape=input_shape)

# CONV2D: 8 filters 4x4, stride of 1, padding 'SAME'
Z1 = tf.keras.layers.Conv2D(8, (4, 4), strides=1, padding='same')(input_img)

# RELU activation
A1 = tf.keras.layers.ReLU()(Z1)

# MAXPOOL: window 8x8, stride 8, padding 'SAME'
P1 = tf.keras.layers.MaxPooling2D(pool_size=(8, 8), strides=8,
→padding='same')(A1)

# CONV2D: 16 filters 2x2, stride 1, padding 'SAME'
Z2 = tf.keras.layers.Conv2D(16, (2, 2), strides=1, padding='same')(P1)

# RELU activation
A2 = tf.keras.layers.ReLU()(Z2)

# MAXPOOL: window 4x4, stride 4, padding 'SAME'
P2 = tf.keras.layers.MaxPooling2D(pool_size=(4, 4), strides=4,
→padding='same')(A2)

# FLATTEN
F = tf.keras.layers.Flatten()(P2)

# Dense layer: 6 neurons, 'softmax' activation for multi-class
→classification
outputs = tf.keras.layers.Dense(6, activation='softmax')(F)
# YOUR CODE ENDS HERE
model = tf.keras.Model(inputs=input_img, outputs=outputs)
return model

```

```

[16]: 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_1 (InputLayer)	[(None, 64, 64, 3)]	0
conv2d_1 (Conv2D)	(None, 64, 64, 8)	392
re_lu_1 (ReLU)	(None, 64, 64, 8)	0
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 8)	0
conv2d_2 (Conv2D)	(None, 8, 8, 16)	528
re_lu_2 (ReLU)	(None, 8, 8, 16)	0
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 16)	0
flatten_1 (Flatten)	(None, 64)	0
dense_1 (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

```
[17]: 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: 1.8089 -
accuracy: 0.1667 - val_loss: 1.7927 - val_accuracy: 0.1750

Epoch 2/100

17/17 [=====] - 2s 106ms/step - loss: 1.7919 -
accuracy: 0.1704 - val_loss: 1.7897 - val_accuracy: 0.1750

Epoch 3/100

17/17 [=====] - 2s 106ms/step - loss: 1.7878 -
accuracy: 0.2352 - val_loss: 1.7860 - val_accuracy: 0.2500

Epoch 4/100

17/17 [=====] - 2s 107ms/step - loss: 1.7814 -
accuracy: 0.2565 - val_loss: 1.7814 - val_accuracy: 0.2500

Epoch 5/100

17/17 [=====] - 2s 106ms/step - loss: 1.7734 -
accuracy: 0.2861 - val_loss: 1.7754 - val_accuracy: 0.2750

Epoch 6/100

17/17 [=====] - 2s 112ms/step - loss: 1.7635 -
accuracy: 0.3306 - val_loss: 1.7677 - val_accuracy: 0.3083

Epoch 7/100

17/17 [=====] - 2s 107ms/step - loss: 1.7498 -
accuracy: 0.3389 - val_loss: 1.7563 - val_accuracy: 0.3167

Epoch 8/100

17/17 [=====] - 2s 106ms/step - loss: 1.7315 -
accuracy: 0.3639 - val_loss: 1.7411 - val_accuracy: 0.3250

Epoch 9/100

17/17 [=====] - 2s 111ms/step - loss: 1.7069 -
accuracy: 0.3630 - val_loss: 1.7232 - val_accuracy: 0.3333

Epoch 10/100

17/17 [=====] - 2s 106ms/step - loss: 1.6721 -
accuracy: 0.3907 - val_loss: 1.6916 - val_accuracy: 0.4083

Epoch 11/100

17/17 [=====] - 2s 106ms/step - loss: 1.6275 -
accuracy: 0.4194 - val_loss: 1.6576 - val_accuracy: 0.4000

Epoch 12/100

17/17 [=====] - 2s 106ms/step - loss: 1.5791 -
accuracy: 0.4426 - val_loss: 1.6199 - val_accuracy: 0.3917

Epoch 13/100

17/17 [=====] - 2s 106ms/step - loss: 1.5289 -
accuracy: 0.4639 - val_loss: 1.5793 - val_accuracy: 0.4083

Epoch 14/100

17/17 [=====] - 2s 111ms/step - loss: 1.4788 -
accuracy: 0.4806 - val_loss: 1.5386 - val_accuracy: 0.4333

Epoch 15/100
17/17 [=====] - 2s 106ms/step - loss: 1.4304 - accuracy: 0.5056 - val_loss: 1.4991 - val_accuracy: 0.4750
Epoch 16/100
17/17 [=====] - 2s 106ms/step - loss: 1.3858 - accuracy: 0.5194 - val_loss: 1.4582 - val_accuracy: 0.4917
Epoch 17/100
17/17 [=====] - 2s 106ms/step - loss: 1.3437 - accuracy: 0.5389 - val_loss: 1.4173 - val_accuracy: 0.5083
Epoch 18/100
17/17 [=====] - 2s 106ms/step - loss: 1.3034 - accuracy: 0.5537 - val_loss: 1.3756 - val_accuracy: 0.5083
Epoch 19/100
17/17 [=====] - 2s 112ms/step - loss: 1.2648 - accuracy: 0.5611 - val_loss: 1.3350 - val_accuracy: 0.5167
Epoch 20/100
17/17 [=====] - 2s 106ms/step - loss: 1.2262 - accuracy: 0.5750 - val_loss: 1.2924 - val_accuracy: 0.5167
Epoch 21/100
17/17 [=====] - 2s 106ms/step - loss: 1.1867 - accuracy: 0.5981 - val_loss: 1.2480 - val_accuracy: 0.5333
Epoch 22/100
17/17 [=====] - 2s 111ms/step - loss: 1.1477 - accuracy: 0.6176 - val_loss: 1.2057 - val_accuracy: 0.5583
Epoch 23/100
17/17 [=====] - 2s 112ms/step - loss: 1.1091 - accuracy: 0.6417 - val_loss: 1.1632 - val_accuracy: 0.5917
Epoch 24/100
17/17 [=====] - 2s 107ms/step - loss: 1.0723 - accuracy: 0.6454 - val_loss: 1.1222 - val_accuracy: 0.6167
Epoch 25/100
17/17 [=====] - 2s 106ms/step - loss: 1.0351 - accuracy: 0.6593 - val_loss: 1.0807 - val_accuracy: 0.6000
Epoch 26/100
17/17 [=====] - 2s 106ms/step - loss: 1.0000 - accuracy: 0.6667 - val_loss: 1.0420 - val_accuracy: 0.6250
Epoch 27/100
17/17 [=====] - 2s 111ms/step - loss: 0.9625 - accuracy: 0.6861 - val_loss: 1.0009 - val_accuracy: 0.6417
Epoch 28/100
17/17 [=====] - 2s 107ms/step - loss: 0.9287 - accuracy: 0.6991 - val_loss: 0.9640 - val_accuracy: 0.6417
Epoch 29/100
17/17 [=====] - 2s 112ms/step - loss: 0.8954 - accuracy: 0.7176 - val_loss: 0.9303 - val_accuracy: 0.6583
Epoch 30/100
17/17 [=====] - 2s 106ms/step - loss: 0.8646 - accuracy: 0.7324 - val_loss: 0.8971 - val_accuracy: 0.6500

Epoch 31/100
17/17 [=====] - 2s 106ms/step - loss: 0.8346 - accuracy: 0.7435 - val_loss: 0.8677 - val_accuracy: 0.6750

Epoch 32/100
17/17 [=====] - 2s 106ms/step - loss: 0.8073 - accuracy: 0.7528 - val_loss: 0.8403 - val_accuracy: 0.6833

Epoch 33/100
17/17 [=====] - 2s 106ms/step - loss: 0.7806 - accuracy: 0.7676 - val_loss: 0.8155 - val_accuracy: 0.6917

Epoch 34/100
17/17 [=====] - 2s 106ms/step - loss: 0.7578 - accuracy: 0.7731 - val_loss: 0.7931 - val_accuracy: 0.7000

Epoch 35/100
17/17 [=====] - 2s 106ms/step - loss: 0.7354 - accuracy: 0.7806 - val_loss: 0.7717 - val_accuracy: 0.7083

Epoch 36/100
17/17 [=====] - 2s 111ms/step - loss: 0.7150 - accuracy: 0.7843 - val_loss: 0.7526 - val_accuracy: 0.7250

Epoch 37/100
17/17 [=====] - 2s 111ms/step - loss: 0.6956 - accuracy: 0.7907 - val_loss: 0.7341 - val_accuracy: 0.7417

Epoch 38/100
17/17 [=====] - 2s 111ms/step - loss: 0.6768 - accuracy: 0.7991 - val_loss: 0.7172 - val_accuracy: 0.7500

Epoch 39/100
17/17 [=====] - 2s 111ms/step - loss: 0.6597 - accuracy: 0.8074 - val_loss: 0.7024 - val_accuracy: 0.7583

Epoch 40/100
17/17 [=====] - 2s 112ms/step - loss: 0.6437 - accuracy: 0.8157 - val_loss: 0.6879 - val_accuracy: 0.7667

Epoch 41/100
17/17 [=====] - 2s 106ms/step - loss: 0.6286 - accuracy: 0.8204 - val_loss: 0.6748 - val_accuracy: 0.7667

Epoch 42/100
17/17 [=====] - 2s 106ms/step - loss: 0.6146 - accuracy: 0.8259 - val_loss: 0.6621 - val_accuracy: 0.7667

Epoch 43/100
17/17 [=====] - 2s 106ms/step - loss: 0.6013 - accuracy: 0.8287 - val_loss: 0.6490 - val_accuracy: 0.7667

Epoch 44/100
17/17 [=====] - 2s 112ms/step - loss: 0.5884 - accuracy: 0.8352 - val_loss: 0.6381 - val_accuracy: 0.7833

Epoch 45/100
17/17 [=====] - 2s 112ms/step - loss: 0.5766 - accuracy: 0.8407 - val_loss: 0.6277 - val_accuracy: 0.7917

Epoch 46/100
17/17 [=====] - 2s 112ms/step - loss: 0.5655 - accuracy: 0.8417 - val_loss: 0.6185 - val_accuracy: 0.8000

Epoch 47/100
17/17 [=====] - 2s 106ms/step - loss: 0.5545 - accuracy: 0.8481 - val_loss: 0.6085 - val_accuracy: 0.8000

Epoch 48/100
17/17 [=====] - 2s 112ms/step - loss: 0.5445 - accuracy: 0.8509 - val_loss: 0.6004 - val_accuracy: 0.8000

Epoch 49/100
17/17 [=====] - 2s 106ms/step - loss: 0.5340 - accuracy: 0.8565 - val_loss: 0.5917 - val_accuracy: 0.8250

Epoch 50/100
17/17 [=====] - 2s 106ms/step - loss: 0.5250 - accuracy: 0.8565 - val_loss: 0.5845 - val_accuracy: 0.8167

Epoch 51/100
17/17 [=====] - 2s 106ms/step - loss: 0.5152 - accuracy: 0.8639 - val_loss: 0.5770 - val_accuracy: 0.8250

Epoch 52/100
17/17 [=====] - 2s 106ms/step - loss: 0.5068 - accuracy: 0.8611 - val_loss: 0.5698 - val_accuracy: 0.8250

Epoch 53/100
17/17 [=====] - 2s 106ms/step - loss: 0.4987 - accuracy: 0.8648 - val_loss: 0.5625 - val_accuracy: 0.8250

Epoch 54/100
17/17 [=====] - 2s 106ms/step - loss: 0.4901 - accuracy: 0.8648 - val_loss: 0.5559 - val_accuracy: 0.8250

Epoch 55/100
17/17 [=====] - 2s 106ms/step - loss: 0.4825 - accuracy: 0.8676 - val_loss: 0.5505 - val_accuracy: 0.8250

Epoch 56/100
17/17 [=====] - 2s 106ms/step - loss: 0.4748 - accuracy: 0.8722 - val_loss: 0.5434 - val_accuracy: 0.8333

Epoch 57/100
17/17 [=====] - 2s 106ms/step - loss: 0.4679 - accuracy: 0.8741 - val_loss: 0.5387 - val_accuracy: 0.8333

Epoch 58/100
17/17 [=====] - 2s 106ms/step - loss: 0.4606 - accuracy: 0.8750 - val_loss: 0.5308 - val_accuracy: 0.8417

Epoch 59/100
17/17 [=====] - 2s 106ms/step - loss: 0.4542 - accuracy: 0.8796 - val_loss: 0.5277 - val_accuracy: 0.8333

Epoch 60/100
17/17 [=====] - 2s 112ms/step - loss: 0.4478 - accuracy: 0.8815 - val_loss: 0.5202 - val_accuracy: 0.8417

Epoch 61/100
17/17 [=====] - 2s 111ms/step - loss: 0.4418 - accuracy: 0.8852 - val_loss: 0.5169 - val_accuracy: 0.8333

Epoch 62/100
17/17 [=====] - 2s 112ms/step - loss: 0.4353 - accuracy: 0.8843 - val_loss: 0.5103 - val_accuracy: 0.8500

Epoch 63/100
17/17 [=====] - 2s 111ms/step - loss: 0.4298 - accuracy: 0.8852 - val_loss: 0.5073 - val_accuracy: 0.8417

Epoch 64/100
17/17 [=====] - 2s 111ms/step - loss: 0.4241 - accuracy: 0.8861 - val_loss: 0.5011 - val_accuracy: 0.8417

Epoch 65/100
17/17 [=====] - 2s 106ms/step - loss: 0.4187 - accuracy: 0.8870 - val_loss: 0.4982 - val_accuracy: 0.8417

Epoch 66/100
17/17 [=====] - 2s 106ms/step - loss: 0.4131 - accuracy: 0.8907 - val_loss: 0.4922 - val_accuracy: 0.8417

Epoch 67/100
17/17 [=====] - 2s 106ms/step - loss: 0.4080 - accuracy: 0.8917 - val_loss: 0.4900 - val_accuracy: 0.8417

Epoch 68/100
17/17 [=====] - 2s 106ms/step - loss: 0.4028 - accuracy: 0.8926 - val_loss: 0.4847 - val_accuracy: 0.8417

Epoch 69/100
17/17 [=====] - 2s 106ms/step - loss: 0.3981 - accuracy: 0.8926 - val_loss: 0.4822 - val_accuracy: 0.8417

Epoch 70/100
17/17 [=====] - 2s 106ms/step - loss: 0.3927 - accuracy: 0.8954 - val_loss: 0.4769 - val_accuracy: 0.8417

Epoch 71/100
17/17 [=====] - 2s 106ms/step - loss: 0.3881 - accuracy: 0.8991 - val_loss: 0.4743 - val_accuracy: 0.8500

Epoch 72/100
17/17 [=====] - 2s 111ms/step - loss: 0.3833 - accuracy: 0.9009 - val_loss: 0.4701 - val_accuracy: 0.8500

Epoch 73/100
17/17 [=====] - 2s 112ms/step - loss: 0.3792 - accuracy: 0.9000 - val_loss: 0.4674 - val_accuracy: 0.8500

Epoch 74/100
17/17 [=====] - 2s 106ms/step - loss: 0.3744 - accuracy: 0.9028 - val_loss: 0.4633 - val_accuracy: 0.8583

Epoch 75/100
17/17 [=====] - 2s 106ms/step - loss: 0.3704 - accuracy: 0.9028 - val_loss: 0.4613 - val_accuracy: 0.8583

Epoch 76/100
17/17 [=====] - 2s 106ms/step - loss: 0.3660 - accuracy: 0.9046 - val_loss: 0.4572 - val_accuracy: 0.8583

Epoch 77/100
17/17 [=====] - 2s 106ms/step - loss: 0.3623 - accuracy: 0.9065 - val_loss: 0.4549 - val_accuracy: 0.8583

Epoch 78/100
17/17 [=====] - 2s 107ms/step - loss: 0.3581 - accuracy: 0.9083 - val_loss: 0.4516 - val_accuracy: 0.8583

Epoch 79/100
17/17 [=====] - 2s 106ms/step - loss: 0.3543 -
accuracy: 0.9083 - val_loss: 0.4487 - val_accuracy: 0.8583
Epoch 80/100
17/17 [=====] - 2s 106ms/step - loss: 0.3504 -
accuracy: 0.9074 - val_loss: 0.4456 - val_accuracy: 0.8583
Epoch 81/100
17/17 [=====] - 2s 106ms/step - loss: 0.3468 -
accuracy: 0.9083 - val_loss: 0.4434 - val_accuracy: 0.8583
Epoch 82/100
17/17 [=====] - 2s 107ms/step - loss: 0.3431 -
accuracy: 0.9093 - val_loss: 0.4404 - val_accuracy: 0.8583
Epoch 83/100
17/17 [=====] - 2s 106ms/step - loss: 0.3398 -
accuracy: 0.9102 - val_loss: 0.4383 - val_accuracy: 0.8583
Epoch 84/100
17/17 [=====] - 2s 107ms/step - loss: 0.3362 -
accuracy: 0.9102 - val_loss: 0.4354 - val_accuracy: 0.8583
Epoch 85/100
17/17 [=====] - 2s 106ms/step - loss: 0.3328 -
accuracy: 0.9139 - val_loss: 0.4335 - val_accuracy: 0.8583
Epoch 86/100
17/17 [=====] - 2s 102ms/step - loss: 0.3295 -
accuracy: 0.9130 - val_loss: 0.4312 - val_accuracy: 0.8583
Epoch 87/100
17/17 [=====] - 2s 106ms/step - loss: 0.3263 -
accuracy: 0.9157 - val_loss: 0.4288 - val_accuracy: 0.8583
Epoch 88/100
17/17 [=====] - 2s 106ms/step - loss: 0.3230 -
accuracy: 0.9148 - val_loss: 0.4266 - val_accuracy: 0.8583
Epoch 89/100
17/17 [=====] - 2s 111ms/step - loss: 0.3199 -
accuracy: 0.9167 - val_loss: 0.4246 - val_accuracy: 0.8583
Epoch 90/100
17/17 [=====] - 2s 111ms/step - loss: 0.3168 -
accuracy: 0.9157 - val_loss: 0.4224 - val_accuracy: 0.8583
Epoch 91/100
17/17 [=====] - 2s 107ms/step - loss: 0.3139 -
accuracy: 0.9167 - val_loss: 0.4204 - val_accuracy: 0.8583
Epoch 92/100
17/17 [=====] - 2s 106ms/step - loss: 0.3109 -
accuracy: 0.9167 - val_loss: 0.4183 - val_accuracy: 0.8583
Epoch 93/100
17/17 [=====] - 2s 106ms/step - loss: 0.3081 -
accuracy: 0.9167 - val_loss: 0.4170 - val_accuracy: 0.8583
Epoch 94/100
17/17 [=====] - 2s 106ms/step - loss: 0.3050 -
accuracy: 0.9167 - val_loss: 0.4144 - val_accuracy: 0.8583

```

Epoch 95/100
17/17 [=====] - 2s 106ms/step - loss: 0.3024 -
accuracy: 0.9167 - val_loss: 0.4129 - val_accuracy: 0.8583
Epoch 96/100
17/17 [=====] - 2s 106ms/step - loss: 0.2994 -
accuracy: 0.9167 - val_loss: 0.4109 - val_accuracy: 0.8583
Epoch 97/100
17/17 [=====] - 2s 106ms/step - loss: 0.2968 -
accuracy: 0.9176 - val_loss: 0.4088 - val_accuracy: 0.8500
Epoch 98/100
17/17 [=====] - 2s 106ms/step - loss: 0.2939 -
accuracy: 0.9204 - val_loss: 0.4071 - val_accuracy: 0.8500
Epoch 99/100
17/17 [=====] - 2s 107ms/step - loss: 0.2915 -
accuracy: 0.9213 - val_loss: 0.4050 - val_accuracy: 0.8417
Epoch 100/100
17/17 [=====] - 2s 107ms/step - loss: 0.2886 -
accuracy: 0.9231 - val_loss: 0.4031 - val_accuracy: 0.8417

```

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`:

```
[18]: history.history
```

```

[18]: {'loss': [1.808906078338623,
1.791914701461792,
1.7877589464187622,
1.7814217805862427,
1.7733564376831055,
1.7635225057601929,
1.7497706413269043,
1.731458067893982,
1.7068579196929932,
1.6720632314682007,
1.6274923086166382,
1.5791007280349731,
1.5288664102554321,
1.4787739515304565,
1.430357813835144,
1.3857589960098267,
1.3436962366104126,
1.3034309148788452,
1.2647631168365479,
1.2262135744094849,
1.1867313385009766,
1.1476670503616333,

```

1.1090983152389526,
1.072251319885254,
1.035092830657959,
0.9999573826789856,
0.9625193476676941,
0.9286597371101379,
0.8953980207443237,
0.8645952939987183,
0.8345773220062256,
0.8073326349258423,
0.7806497812271118,
0.7577588558197021,
0.7354241609573364,
0.7150248289108276,
0.6956374049186707,
0.6767770051956177,
0.6596775650978088,
0.6437271237373352,
0.6285879611968994,
0.6146488189697266,
0.6012987494468689,
0.5884119272232056,
0.5765707492828369,
0.5654748678207397,
0.554487407207489,
0.5445135831832886,
0.534043550491333,
0.5249912738800049,
0.5152434706687927,
0.5068246126174927,
0.49865463376045227,
0.4901266098022461,
0.48251861333847046,
0.4748465120792389,
0.46787410974502563,
0.4606393873691559,
0.4541715383529663,
0.44776880741119385,
0.44180962443351746,
0.43531185388565063,
0.42981529235839844,
0.42414984107017517,
0.4187040627002716,
0.41310665011405945,
0.40798866748809814,
0.40277183055877686,
0.39806392788887024,

0.39268478751182556,
0.38807737827301025,
0.3832712769508362,
0.3791556656360626,
0.37444114685058594,
0.37043672800064087,
0.3660280406475067,
0.36228668689727783,
0.35806334018707275,
0.3543021082878113,
0.350398451089859,
0.34675055742263794,
0.34306639432907104,
0.3398013114929199,
0.3362336754798889,
0.332804411649704,
0.3295277953147888,
0.3263339698314667,
0.32296207547187805,
0.31992632150650024,
0.3168497085571289,
0.3138992190361023,
0.31087881326675415,
0.30807730555534363,
0.3050267994403839,
0.30242881178855896,
0.2993841767311096,
0.2968268096446991,
0.29390546679496765,
0.29148420691490173,
0.2885875701904297],
'accuracy': [0.1666666716337204,
0.17037037014961243,
0.2351851910352707,
0.25648146867752075,
0.28611111640930176,
0.3305555582046509,
0.33888888359069824,
0.3638888895511627,
0.3629629611968994,
0.3907407522201538,
0.4194444417953491,
0.442592591047287,
0.46388888359069824,
0.48055556416511536,
0.5055555701255798,
0.519444465637207,

0.5388888716697693,
0.5537037253379822,
0.5611110925674438,
0.574999988079071,
0.5981481671333313,
0.6175925731658936,
0.6416666507720947,
0.645370364189148,
0.6592592597007751,
0.6666666865348816,
0.6861110925674438,
0.6990740895271301,
0.7175925970077515,
0.7324073910713196,
0.7435185313224792,
0.7527777552604675,
0.7675926089286804,
0.7731481194496155,
0.780555462837219,
0.7842592597007751,
0.7907407283782959,
0.7990740537643433,
0.8074073791503906,
0.8157407641410828,
0.8203703761100769,
0.8259259462356567,
0.8287037014961243,
0.835185170173645,
0.8407407402992249,
0.8416666388511658,
0.8481481671333313,
0.8509259223937988,
0.8564814925193787,
0.8564814925193787,
0.8638888597488403,
0.8611111044883728,
0.864814817905426,
0.864814817905426,
0.8675925731658936,
0.8722222447395325,
0.8740741014480591,
0.875,
0.8796296119689941,
0.8814814686775208,
0.885185182094574,
0.8842592835426331,
0.885185182094574,

0.8861111402511597,
0.8870370388031006,
0.8907407522201538,
0.8916666507720947,
0.8925926089286804,
0.8925926089286804,
0.895370364189148,
0.8990740776062012,
0.9009259343147278,
0.8999999761581421,
0.9027777910232544,
0.9027777910232544,
0.904629647731781,
0.9064815044403076,
0.9083333611488342,
0.9083333611488342,
0.9074074029922485,
0.9083333611488342,
0.9092592597007751,
0.9101851582527161,
0.9101851582527161,
0.9138888716697693,
0.9129629731178284,
0.9157407283782959,
0.914814829826355,
0.9166666865348816,
0.9157407283782959,
0.9166666865348816,
0.9166666865348816,
0.9166666865348816,
0.9166666865348816,
0.9166666865348816,
0.9166666865348816,
0.9166666865348816,
0.9175925850868225,
0.9203703999519348,
0.9212962985038757,
0.9231481552124023],
'val_loss': [1.792737364768982,
1.7897276878356934,
1.7860122919082642,
1.781379222869873,
1.775438904762268,
1.7677334547042847,
1.7563166618347168,
1.7411319017410278,
1.7232491970062256,
1.6916046142578125,

1.6575572490692139,
1.6198757886886597,
1.5793437957763672,
1.538615107536316,
1.4990869760513306,
1.4581960439682007,
1.4172605276107788,
1.3756085634231567,
1.3350282907485962,
1.2923598289489746,
1.2479907274246216,
1.2057030200958252,
1.1632388830184937,
1.122244119644165,
1.080675721168518,
1.0419751405715942,
1.000915765762329,
0.9640265107154846,
0.9303340911865234,
0.8971179723739624,
0.8676890134811401,
0.8403358459472656,
0.8154626488685608,
0.7930517792701721,
0.7716618180274963,
0.7526119947433472,
0.7341195344924927,
0.7171943783760071,
0.7023608088493347,
0.6879228949546814,
0.6747750043869019,
0.6620591878890991,
0.6489793062210083,
0.6381200551986694,
0.6276541352272034,
0.6184623837471008,
0.608490526676178,
0.6004272699356079,
0.5916524529457092,
0.5845184326171875,
0.5770260095596313,
0.5697553753852844,
0.5624591708183289,
0.5558763742446899,
0.5505078434944153,
0.5433782339096069,
0.5387219786643982,

0.5307773947715759,
0.5276778936386108,
0.5201903581619263,
0.516906201839447,
0.5102750658988953,
0.5072909593582153,
0.5010647177696228,
0.49820297956466675,
0.492177277803421,
0.4900434613227844,
0.484674870967865,
0.48221126198768616,
0.4769119918346405,
0.47428378462791443,
0.4701434373855591,
0.46743375062942505,
0.4633057415485382,
0.4612957835197449,
0.45715439319610596,
0.45486146211624146,
0.451640248298645,
0.4487344026565552,
0.445562481880188,
0.44343870878219604,
0.44044533371925354,
0.4382862150669098,
0.4353589415550232,
0.433514267206192,
0.43117618560791016,
0.42880114912986755,
0.42659395933151245,
0.4245795011520386,
0.42235735058784485,
0.42044857144355774,
0.41827595233917236,
0.41702836751937866,
0.41438040137290955,
0.4128895401954651,
0.4109305143356323,
0.40880918502807617,
0.40714818239212036,
0.40497729182243347,
0.40307852625846863],
'val_accuracy': [0.17499999701976776,
0.17499999701976776,
0.25,
0.25,

0.2750000059604645,
0.3083333373069763,
0.3166666626930237,
0.32499998807907104,
0.3333333432674408,
0.40833333134651184,
0.4000000059604645,
0.3916666805744171,
0.40833333134651184,
0.4333333373069763,
0.4749999940395355,
0.49166667461395264,
0.5083333253860474,
0.5083333253860474,
0.5166666507720947,
0.5166666507720947,
0.5333333611488342,
0.5583333373069763,
0.5916666388511658,
0.6166666746139526,
0.6000000238418579,
0.625,
0.6416666507720947,
0.6416666507720947,
0.6583333611488342,
0.6499999761581421,
0.675000011920929,
0.6833333373069763,
0.6916666626930237,
0.699999988079071,
0.7083333134651184,
0.7250000238418579,
0.7416666746139526,
0.75,
0.7583333253860474,
0.7666666507720947,
0.7666666507720947,
0.7666666507720947,
0.7666666507720947,
0.7833333611488342,
0.7916666865348816,
0.800000011920929,
0.800000011920929,
0.800000011920929,
0.824999988079071,
0.8166666626930237,
0.824999988079071,

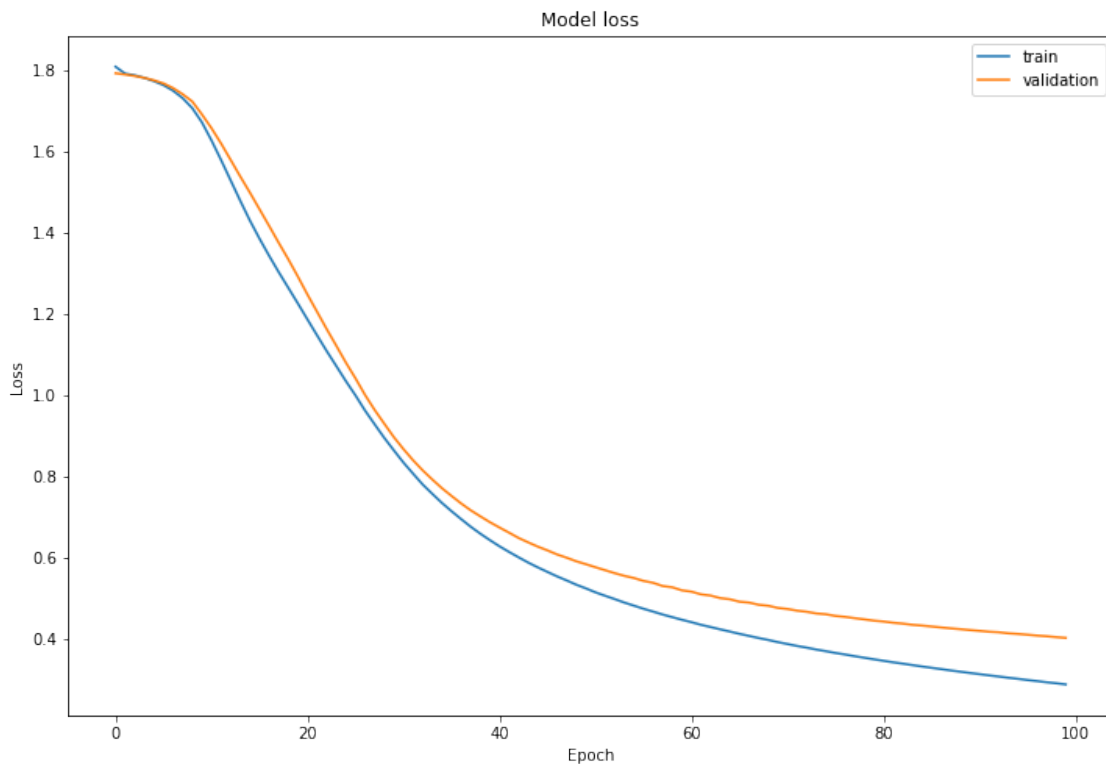
[illegible]

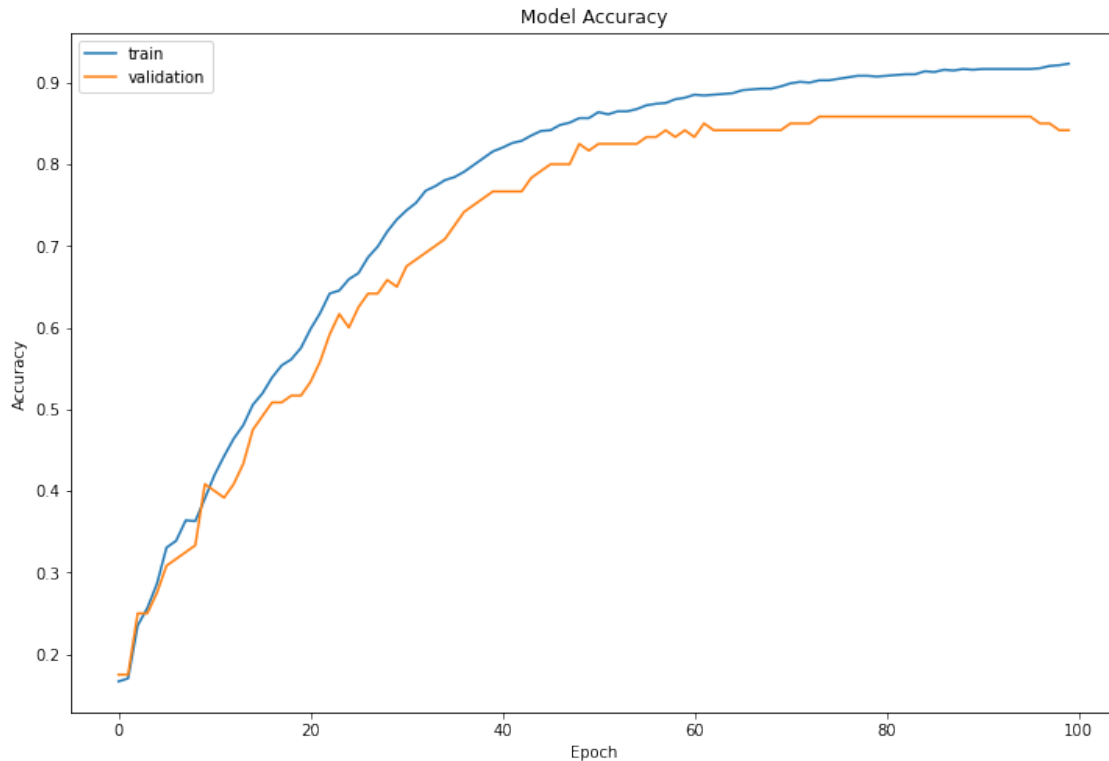
```
0.8416666388511658,  
0.8416666388511658]]}
```

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

```
[19]: # 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')
```

```
[19]: [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>