Convolution_model_Application

July 13, 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.

```
[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))
```
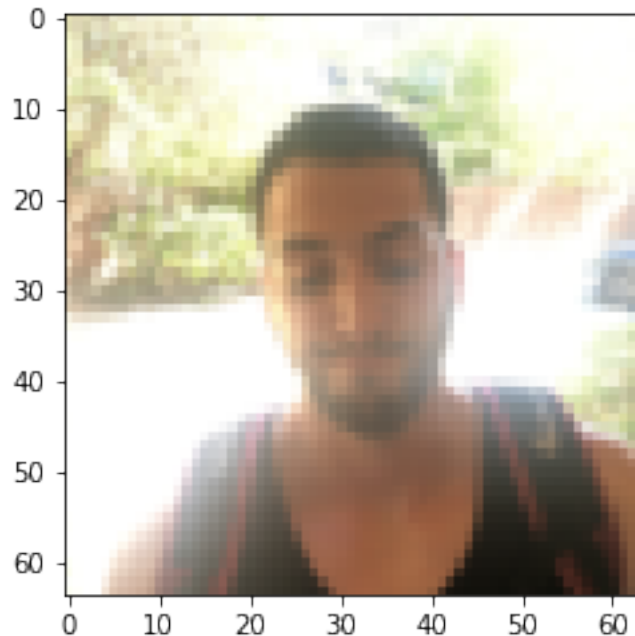
```
number of training examples = 600
number of test examples = 150
X_train shape: (600, 64, 64, 3)
Y_train shape: (600, 1)
X_test shape: (150, 64, 64, 3)
Y_test shape: (150, 1)
```

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**

```
# 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
```

```python
    Returns:
    model -- TF Keras model (object containing the information for the entire␣
  ↪training process)
    """
    model = tf.keras.Sequential([
            ## ZeroPadding2D with padding 3, input shape of 64 x 64 x 3
            tfl.ZeroPadding2D(padding=(3,3), input_shape = (64, 64, 3)),

            ## Conv2D with 32 7x7 filters and stride of 1
            tfl.Conv2D(filters=32, kernel_size=7),

            ## BatchNormalization for axis 3
            tfl.BatchNormalization(axis=3),

            ## ReLU
            tfl.ReLU(),

            ## Max Pooling 2D with default parameters
            tfl.MaxPool2D(),

            ## Flatten layer
            tfl.Flatten(),

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

            # YOUR CODE STARTS HERE
            tfl.Dense(units=1, activation="sigmoid")

            # YOUR CODE ENDS HERE
        ])

    return model
```

```python
[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!
```

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 (ZeroPadding2 (None, 70, 70, 3)         0

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

_____
batch_normalization (BatchNo (None, 64, 64, 32)        128

_____
re_lu (ReLU)                 (None, 64, 64, 32)        0

_____
max_pooling2d (MaxPooling2D) (None, 32, 32, 32)        0

_____
flatten (Flatten)            (None, 32768)             0

_____
dense (Dense)                (None, 1)                 32769
=================================================================
Total params: 37,633
Trainable params: 37,569
Non-trainable params: 64
```

3.2 - Train and Evaluate the Model

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

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

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

```
Epoch 1/10
38/38 [==============================] - 4s 103ms/step - loss: 1.3015 -
accuracy: 0.6600
Epoch 2/10
38/38 [==============================] - 4s 97ms/step - loss: 0.2405 - accuracy:
0.8950
Epoch 3/10
38/38 [==============================] - 4s 97ms/step - loss: 0.1968 - accuracy:
0.9283
Epoch 4/10
38/38 [==============================] - 4s 100ms/step - loss: 0.1628 -
accuracy: 0.9383
Epoch 5/10
38/38 [==============================] - 4s 95ms/step - loss: 0.1402 - accuracy:
0.9417
Epoch 6/10
38/38 [==============================] - 4s 95ms/step - loss: 0.1309 - accuracy:
0.9483
Epoch 7/10
38/38 [==============================] - 4s 97ms/step - loss: 0.0836 - accuracy:
0.9683
Epoch 8/10
38/38 [==============================] - 4s 95ms/step - loss: 0.0615 - accuracy:
0.9767
Epoch 9/10
38/38 [==============================] - 4s 97ms/step - loss: 0.0537 - accuracy:
0.9817
Epoch 10/10
38/38 [==============================] - 4s 95ms/step - loss: 0.0533 - accuracy:
0.9833
```

```
[8]: <tensorflow.python.keras.callbacks.History at 0x7f017738fe10>
```

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 34ms/step - loss: 0.1879 - accuracy:
0.9067
```

```
[9]:  [0.18786382675170898, 0.9066666960716248]
```

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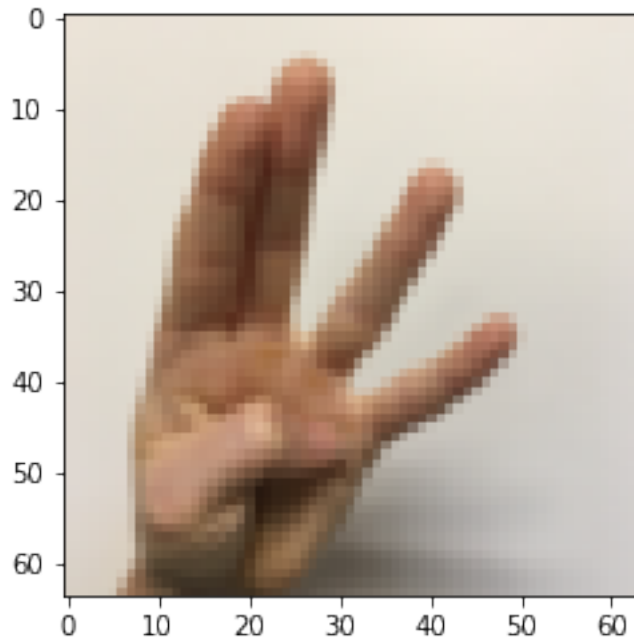
4.2 - Split the Data into Train/Test Sets

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)
    """
    # YOUR CODE STARTS HERE
    input_img = tf.keras.Input(shape=input_shape)
    ## CONV2D: 8 filters 4x4, stride of 1, padding 'SAME'
    Z1 = tfl.Conv2D(filters=8, kernel_size=4, padding='same')(input_img)
    ## RELU
    A1 = tfl.ReLU()(Z1)
    ## MAXPOOL: window 8x8, stride 8, padding 'SAME'
    P1 = tfl.MaxPool2D(pool_size=(8, 8), strides=(8, 8), padding='same')(A1)
    ## CONV2D: 16 filters 2x2, stride 1, padding 'SAME'
    Z2 = tfl.Conv2D(filters=16, kernel_size=2, padding='same')(P1)
```

```
        ## RELU
        A2 = tfl.ReLU()(Z2)
        ## MAXPOOL: window 4x4, stride 4, padding 'SAME'
        P2 = tfl.MaxPool2D(pool_size=(4, 4), strides=(4, 4), padding='same')(A2)
        ## FLATTEN
        F = tfl.Flatten()(P2)
        ## Dense layer
        ## 6 neurons in output layer. Hint: one of the arguments should be␣
    ↪"activation='softmax'"
        outputs = tfl.Dense(units=6, activation='softmax')(F)

        # YOUR CODE 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_2 (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_2 (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

```python
[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 106ms/step - loss: 1.8030 -
accuracy: 0.1778 - val_loss: 1.7888 - val_accuracy: 0.2000
Epoch 2/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7866 -
accuracy: 0.1880 - val_loss: 1.7848 - val_accuracy: 0.1833
Epoch 3/100
17/17 [==============================] - 2s 107ms/step - loss: 1.7809 -
accuracy: 0.1713 - val_loss: 1.7803 - val_accuracy: 0.1833
Epoch 4/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7754 -
accuracy: 0.1963 - val_loss: 1.7755 - val_accuracy: 0.2333
Epoch 5/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7696 -
accuracy: 0.2435 - val_loss: 1.7700 - val_accuracy: 0.2667
Epoch 6/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7618 -
accuracy: 0.2750 - val_loss: 1.7624 - val_accuracy: 0.2833
Epoch 7/100
17/17 [==============================] - 2s 107ms/step - loss: 1.7519 -
accuracy: 0.3259 - val_loss: 1.7521 - val_accuracy: 0.3500
Epoch 8/100
17/17 [==============================] - 2s 106ms/step - loss: 1.7385 -
accuracy: 0.3620 - val_loss: 1.7384 - val_accuracy: 0.3667
Epoch 9/100
```

13

```
17/17 [==============================] - 2s 111ms/step - loss: 1.7210 -
accuracy: 0.3981 - val_loss: 1.7216 - val_accuracy: 0.4000
Epoch 10/100
17/17 [==============================] - 2s 106ms/step - loss: 1.6980 -
accuracy: 0.4111 - val_loss: 1.6992 - val_accuracy: 0.4000
Epoch 11/100
17/17 [==============================] - 2s 106ms/step - loss: 1.6690 -
accuracy: 0.4361 - val_loss: 1.6701 - val_accuracy: 0.4000
Epoch 12/100
17/17 [==============================] - 2s 107ms/step - loss: 1.6331 -
accuracy: 0.4500 - val_loss: 1.6355 - val_accuracy: 0.4167
Epoch 13/100
17/17 [==============================] - 2s 106ms/step - loss: 1.5906 -
accuracy: 0.4620 - val_loss: 1.5952 - val_accuracy: 0.4250
Epoch 14/100
17/17 [==============================] - 2s 106ms/step - loss: 1.5422 -
accuracy: 0.4593 - val_loss: 1.5550 - val_accuracy: 0.4750
Epoch 15/100
17/17 [==============================] - 2s 106ms/step - loss: 1.4932 -
accuracy: 0.4852 - val_loss: 1.5137 - val_accuracy: 0.4583
Epoch 16/100
17/17 [==============================] - 2s 106ms/step - loss: 1.4442 -
accuracy: 0.4870 - val_loss: 1.4733 - val_accuracy: 0.4583
Epoch 17/100
17/17 [==============================] - 2s 106ms/step - loss: 1.3978 -
accuracy: 0.4926 - val_loss: 1.4271 - val_accuracy: 0.4833
Epoch 18/100
17/17 [==============================] - 2s 107ms/step - loss: 1.3542 -
accuracy: 0.5111 - val_loss: 1.3893 - val_accuracy: 0.5000
Epoch 19/100
17/17 [==============================] - 2s 106ms/step - loss: 1.3114 -
accuracy: 0.5324 - val_loss: 1.3462 - val_accuracy: 0.5083
Epoch 20/100
17/17 [==============================] - 2s 106ms/step - loss: 1.2725 -
accuracy: 0.5472 - val_loss: 1.3086 - val_accuracy: 0.5083
Epoch 21/100
17/17 [==============================] - 2s 106ms/step - loss: 1.2347 -
accuracy: 0.5630 - val_loss: 1.2699 - val_accuracy: 0.5500
Epoch 22/100
17/17 [==============================] - 2s 106ms/step - loss: 1.1996 -
accuracy: 0.5731 - val_loss: 1.2362 - val_accuracy: 0.5750
Epoch 23/100
17/17 [==============================] - 2s 106ms/step - loss: 1.1662 -
accuracy: 0.5889 - val_loss: 1.2006 - val_accuracy: 0.6250
Epoch 24/100
17/17 [==============================] - 2s 107ms/step - loss: 1.1337 -
accuracy: 0.6130 - val_loss: 1.1663 - val_accuracy: 0.6250
Epoch 25/100
```

```
17/17 [==============================] - 2s 106ms/step - loss: 1.1017 -
accuracy: 0.6269 - val_loss: 1.1323 - val_accuracy: 0.6500
Epoch 26/100
17/17 [==============================] - 2s 106ms/step - loss: 1.0717 -
accuracy: 0.6417 - val_loss: 1.1027 - val_accuracy: 0.6417
Epoch 27/100
17/17 [==============================] - 2s 106ms/step - loss: 1.0425 -
accuracy: 0.6454 - val_loss: 1.0738 - val_accuracy: 0.6417
Epoch 28/100
17/17 [==============================] - 2s 106ms/step - loss: 1.0153 -
accuracy: 0.6611 - val_loss: 1.0482 - val_accuracy: 0.6583
Epoch 29/100
17/17 [==============================] - 2s 106ms/step - loss: 0.9895 -
accuracy: 0.6713 - val_loss: 1.0243 - val_accuracy: 0.6667
Epoch 30/100
17/17 [==============================] - 2s 106ms/step - loss: 0.9644 -
accuracy: 0.6824 - val_loss: 1.0001 - val_accuracy: 0.6917
Epoch 31/100
17/17 [==============================] - 2s 106ms/step - loss: 0.9407 -
accuracy: 0.6889 - val_loss: 0.9799 - val_accuracy: 0.7083
Epoch 32/100
17/17 [==============================] - 2s 112ms/step - loss: 0.9195 -
accuracy: 0.6935 - val_loss: 0.9611 - val_accuracy: 0.7000
Epoch 33/100
17/17 [==============================] - 2s 106ms/step - loss: 0.8983 -
accuracy: 0.7074 - val_loss: 0.9450 - val_accuracy: 0.6917
Epoch 34/100
17/17 [==============================] - 2s 106ms/step - loss: 0.8787 -
accuracy: 0.7093 - val_loss: 0.9280 - val_accuracy: 0.7250
Epoch 35/100
17/17 [==============================] - 2s 106ms/step - loss: 0.8604 -
accuracy: 0.7185 - val_loss: 0.9137 - val_accuracy: 0.7083
Epoch 36/100
17/17 [==============================] - 2s 101ms/step - loss: 0.8434 -
accuracy: 0.7194 - val_loss: 0.8993 - val_accuracy: 0.7250
Epoch 37/100
17/17 [==============================] - 2s 106ms/step - loss: 0.8283 -
accuracy: 0.7315 - val_loss: 0.8908 - val_accuracy: 0.7250
Epoch 38/100
17/17 [==============================] - 2s 111ms/step - loss: 0.8137 -
accuracy: 0.7287 - val_loss: 0.8789 - val_accuracy: 0.7083
Epoch 39/100
17/17 [==============================] - 2s 106ms/step - loss: 0.8004 -
accuracy: 0.7361 - val_loss: 0.8698 - val_accuracy: 0.7250
Epoch 40/100
17/17 [==============================] - 2s 106ms/step - loss: 0.7868 -
accuracy: 0.7380 - val_loss: 0.8557 - val_accuracy: 0.7167
Epoch 41/100
```

```
17/17 [==============================] - 2s 106ms/step - loss: 0.7722 -
accuracy: 0.7454 - val_loss: 0.8436 - val_accuracy: 0.7167
Epoch 42/100
17/17 [==============================] - 2s 106ms/step - loss: 0.7592 -
accuracy: 0.7472 - val_loss: 0.8326 - val_accuracy: 0.7167
Epoch 43/100
17/17 [==============================] - 2s 111ms/step - loss: 0.7474 -
accuracy: 0.7472 - val_loss: 0.8262 - val_accuracy: 0.7167
Epoch 44/100
17/17 [==============================] - 2s 106ms/step - loss: 0.7364 -
accuracy: 0.7565 - val_loss: 0.8145 - val_accuracy: 0.7083
Epoch 45/100
17/17 [==============================] - 2s 106ms/step - loss: 0.7248 -
accuracy: 0.7593 - val_loss: 0.8080 - val_accuracy: 0.7083
Epoch 46/100
17/17 [==============================] - 2s 106ms/step - loss: 0.7147 -
accuracy: 0.7611 - val_loss: 0.7973 - val_accuracy: 0.7167
Epoch 47/100
17/17 [==============================] - 2s 111ms/step - loss: 0.7044 -
accuracy: 0.7667 - val_loss: 0.7896 - val_accuracy: 0.7083
Epoch 48/100
17/17 [==============================] - 2s 107ms/step - loss: 0.6940 -
accuracy: 0.7713 - val_loss: 0.7816 - val_accuracy: 0.7167
Epoch 49/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6851 -
accuracy: 0.7685 - val_loss: 0.7740 - val_accuracy: 0.7250
Epoch 50/100
17/17 [==============================] - 2s 101ms/step - loss: 0.6777 -
accuracy: 0.7722 - val_loss: 0.7665 - val_accuracy: 0.7167
Epoch 51/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6684 -
accuracy: 0.7778 - val_loss: 0.7600 - val_accuracy: 0.7167
Epoch 52/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6593 -
accuracy: 0.7861 - val_loss: 0.7539 - val_accuracy: 0.7250
Epoch 53/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6504 -
accuracy: 0.7852 - val_loss: 0.7477 - val_accuracy: 0.7250
Epoch 54/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6420 -
accuracy: 0.7870 - val_loss: 0.7437 - val_accuracy: 0.7250
Epoch 55/100
17/17 [==============================] - 2s 111ms/step - loss: 0.6339 -
accuracy: 0.7889 - val_loss: 0.7376 - val_accuracy: 0.7250
Epoch 56/100
17/17 [==============================] - 2s 107ms/step - loss: 0.6264 -
accuracy: 0.7907 - val_loss: 0.7319 - val_accuracy: 0.7250
Epoch 57/100
```

```
17/17 [==============================] - 2s 106ms/step - loss: 0.6192 -
accuracy: 0.7954 - val_loss: 0.7259 - val_accuracy: 0.7250
Epoch 58/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6116 -
accuracy: 0.7991 - val_loss: 0.7201 - val_accuracy: 0.7333
Epoch 59/100
17/17 [==============================] - 2s 106ms/step - loss: 0.6049 -
accuracy: 0.8028 - val_loss: 0.7149 - val_accuracy: 0.7333
Epoch 60/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5973 -
accuracy: 0.8037 - val_loss: 0.7089 - val_accuracy: 0.7250
Epoch 61/100
17/17 [==============================] - 2s 111ms/step - loss: 0.5907 -
accuracy: 0.8046 - val_loss: 0.7036 - val_accuracy: 0.7417
Epoch 62/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5839 -
accuracy: 0.8065 - val_loss: 0.6982 - val_accuracy: 0.7417
Epoch 63/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5774 -
accuracy: 0.8102 - val_loss: 0.6929 - val_accuracy: 0.7500
Epoch 64/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5708 -
accuracy: 0.8120 - val_loss: 0.6882 - val_accuracy: 0.7500
Epoch 65/100
17/17 [==============================] - 2s 107ms/step - loss: 0.5640 -
accuracy: 0.8130 - val_loss: 0.6839 - val_accuracy: 0.7500
Epoch 66/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5580 -
accuracy: 0.8139 - val_loss: 0.6793 - val_accuracy: 0.7583
Epoch 67/100
17/17 [==============================] - 2s 111ms/step - loss: 0.5521 -
accuracy: 0.8176 - val_loss: 0.6750 - val_accuracy: 0.7583
Epoch 68/100
17/17 [==============================] - 2s 111ms/step - loss: 0.5462 -
accuracy: 0.8204 - val_loss: 0.6707 - val_accuracy: 0.7667
Epoch 69/100
17/17 [==============================] - 2s 107ms/step - loss: 0.5403 -
accuracy: 0.8213 - val_loss: 0.6658 - val_accuracy: 0.7667
Epoch 70/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5347 -
accuracy: 0.8204 - val_loss: 0.6614 - val_accuracy: 0.7667
Epoch 71/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5291 -
accuracy: 0.8241 - val_loss: 0.6578 - val_accuracy: 0.7667
Epoch 72/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5238 -
accuracy: 0.8250 - val_loss: 0.6525 - val_accuracy: 0.7750
Epoch 73/100
```

```
17/17 [==============================] - 2s 112ms/step - loss: 0.5186 -
accuracy: 0.8259 - val_loss: 0.6495 - val_accuracy: 0.7750
Epoch 74/100
17/17 [==============================] - 2s 106ms/step - loss: 0.5132 -
accuracy: 0.8250 - val_loss: 0.6459 - val_accuracy: 0.7750
Epoch 75/100
17/17 [==============================] - 2s 111ms/step - loss: 0.5080 -
accuracy: 0.8306 - val_loss: 0.6426 - val_accuracy: 0.7750
Epoch 76/100
17/17 [==============================] - 2s 107ms/step - loss: 0.5033 -
accuracy: 0.8324 - val_loss: 0.6386 - val_accuracy: 0.7833
Epoch 77/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4986 -
accuracy: 0.8352 - val_loss: 0.6347 - val_accuracy: 0.7833
Epoch 78/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4938 -
accuracy: 0.8370 - val_loss: 0.6313 - val_accuracy: 0.7833
Epoch 79/100
17/17 [==============================] - 2s 111ms/step - loss: 0.4891 -
accuracy: 0.8389 - val_loss: 0.6275 - val_accuracy: 0.7833
Epoch 80/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4846 -
accuracy: 0.8444 - val_loss: 0.6250 - val_accuracy: 0.7667
Epoch 81/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4802 -
accuracy: 0.8444 - val_loss: 0.6212 - val_accuracy: 0.7833
Epoch 82/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4756 -
accuracy: 0.8481 - val_loss: 0.6183 - val_accuracy: 0.7750
Epoch 83/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4712 -
accuracy: 0.8491 - val_loss: 0.6145 - val_accuracy: 0.7667
Epoch 84/100
17/17 [==============================] - 2s 111ms/step - loss: 0.4667 -
accuracy: 0.8509 - val_loss: 0.6116 - val_accuracy: 0.7667
Epoch 85/100
17/17 [==============================] - 2s 107ms/step - loss: 0.4626 -
accuracy: 0.8519 - val_loss: 0.6086 - val_accuracy: 0.7667
Epoch 86/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4584 -
accuracy: 0.8556 - val_loss: 0.6058 - val_accuracy: 0.7667
Epoch 87/100
17/17 [==============================] - 2s 111ms/step - loss: 0.4542 -
accuracy: 0.8593 - val_loss: 0.6029 - val_accuracy: 0.7667
Epoch 88/100
17/17 [==============================] - 2s 111ms/step - loss: 0.4499 -
accuracy: 0.8611 - val_loss: 0.5997 - val_accuracy: 0.7667
Epoch 89/100
```

```
17/17 [==============================] - 2s 106ms/step - loss: 0.4459 -
accuracy: 0.8611 - val_loss: 0.5972 - val_accuracy: 0.7667
Epoch 90/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4416 -
accuracy: 0.8620 - val_loss: 0.5938 - val_accuracy: 0.7667
Epoch 91/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4377 -
accuracy: 0.8620 - val_loss: 0.5913 - val_accuracy: 0.7667
Epoch 92/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4338 -
accuracy: 0.8639 - val_loss: 0.5885 - val_accuracy: 0.7667
Epoch 93/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4300 -
accuracy: 0.8639 - val_loss: 0.5852 - val_accuracy: 0.7667
Epoch 94/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4262 -
accuracy: 0.8639 - val_loss: 0.5832 - val_accuracy: 0.7750
Epoch 95/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4227 -
accuracy: 0.8648 - val_loss: 0.5802 - val_accuracy: 0.7750
Epoch 96/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4190 -
accuracy: 0.8667 - val_loss: 0.5780 - val_accuracy: 0.7750
Epoch 97/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4154 -
accuracy: 0.8676 - val_loss: 0.5741 - val_accuracy: 0.7833
Epoch 98/100
17/17 [==============================] - 2s 107ms/step - loss: 0.4117 -
accuracy: 0.8694 - val_loss: 0.5727 - val_accuracy: 0.7750
Epoch 99/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4082 -
accuracy: 0.8704 - val_loss: 0.5692 - val_accuracy: 0.7833
Epoch 100/100
17/17 [==============================] - 2s 106ms/step - loss: 0.4044 -
accuracy: 0.8722 - val_loss: 0.5682 - val_accuracy: 0.7917
```

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': [1.8029652833938599,
    1.786616325378418,
    1.7808558940887451,
    1.7754210233688354,
    1.7696326971054077,
```

1.7618348598480225,
1.7518631219863892,
1.7384806871414185,
1.7209527492523193,
1.6980352401733398,
1.6689727306365967,
1.6331275701522827,
1.590607762336731,
1.5421602725982666,
1.4932215213775635,
1.4442437887191772,
1.397763967514038,
1.3542062044143677,
1.3114078044891357,
1.2725144624710083,
1.2347065210342407,
1.199554681777954,
1.1662414073944092,
1.1337366104125977,
1.1017370223999023,
1.0717452764511108,
1.0425078868865967,
1.015257716178894,
0.9895094037055969,
0.964388370513916,
0.9407046437263489,
0.9194647073745728,
0.8983483910560608,
0.8787124156951904,
0.8604427576065063,
0.8434093594551086,
0.8283044695854187,
0.8136547803878784,
0.8003730773925781,
0.7868216633796692,
0.7721936106681824,
0.7592237591743469,
0.7473630309104919,
0.7363973259925842,
0.724838376045227,
0.7147151231765747,
0.704437255859375,
0.6939780712127686,
0.685065746307373,
0.6777006983757019,
0.6684014201164246,
0.6593406796455383,

0.6504244208335876,
0.6420130133628845,
0.6338841915130615,
0.6264318227767944,
0.619247317314148,
0.6116111874580383,
0.6048855781555176,
0.597269594669342,
0.590747058391571,
0.5839198231697083,
0.5774203538894653,
0.5708059072494507,
0.5640367865562439,
0.5580343008041382,
0.5521093010902405,
0.5462426543235779,
0.5402778387069702,
0.5347146391868591,
0.5290753841400146,
0.5237793326377869,
0.5185996294021606,
0.5132080912590027,
0.5080444812774658,
0.5033087134361267,
0.49858352541923523,
0.4937592148780823,
0.4890897274017334,
0.4846186935901642,
0.4802083373069763,
0.47556695342063904,
0.47120124101638794,
0.4666684567928314,
0.46258580684661865,
0.4583539068698883,
0.4542103111743927,
0.44989627599716187,
0.4459185004234314,
0.44161495566368103,
0.4376767575740814,
0.4337840974330902,
0.4300220012664795,
0.42620548605918884,
0.4226911664009094,
0.41895201802253723,
0.41535815596580505,
0.4116925299167633,
0.40822193026542664,

```
    0.40444403886795044],
'accuracy': [0.17777778208255768,
 0.18796296417713165,
 0.17129629850387573,
 0.19629628956317902,
 0.24351851642131805,
 0.2750000059604645,
 0.32592591643333435,
 0.3620370328426361,
 0.39814814925193787,
 0.41111111640930176,
 0.43611112236976624,
 0.44999998807907104,
 0.46203702688217163,
 0.4592592716217041,
 0.4851851761341095,
 0.4870370328426361,
 0.49259260296821594,
 0.5111111402511597,
 0.5324074029922485,
 0.5472221970558167,
 0.5629629492759705,
 0.5731481313705444,
 0.5888888835906982,
 0.6129629611968994,
 0.6268518567085266,
 0.6416666507720947,
 0.645370364189148,
 0.6611111164093018,
 0.6712962985038757,
 0.6824073791503906,
 0.6888889074325562,
 0.6935185194015503,
 0.7074074149131775,
 0.7092592716217041,
 0.7185184955596924,
 0.7194444537162781,
 0.7314814925193787,
 0.7287036776542664,
 0.7361111044883728,
 0.7379629611968994,
 0.7453703880310059,
 0.7472222447395325,
 0.7472222447395325,
 0.7564814686775208,
 0.7592592835426331,
 0.7611111402511597,
```

0.7666666507720947,
0.7712963223457336,
0.7685185074806213,
0.7722222208976746,
0.7777777910232544,
0.7861111164093018,
0.7851851582527161,
0.7870370149612427,
0.7888888716697693,
0.7907407283782959,
0.7953703999519348,
0.7990740537643433,
0.8027777671813965,
0.8037037253379822,
0.8046296238899231,
0.8064814805984497,
0.8101851940155029,
0.8120370507240295,
0.8129629492759705,
0.8138889074325562,
0.8175926208496094,
0.8203703761100769,
0.8212962746620178,
0.8203703761100769,
0.8240740895271301,
0.824999988079071,
0.8259259462356567,
0.824999988079071,
0.8305555582046509,
0.8324074149131775,
0.835185170173645,
0.8370370268821716,
0.8388888835906982,
0.8444444537162781,
0.8444444537162781,
0.8481481671333313,
0.8490740656852722,
0.8509259223937988,
0.8518518805503845,
0.855555534362793,
0.8592592477798462,
0.8611111044883728,
0.8611111044883728,
0.8620370626449585,
0.8620370626449585,
0.8638888597488403,
0.8638888597488403,

```
 0.8638888597488403,
 0.864814817905426,
 0.8666666746139526,
 0.8675925731658936,
 0.8694444298744202,
 0.8703703880310059,
 0.8722222447395325],
'val_loss': [1.7887529134750366,
 1.784794569015503,
 1.7802797555923462,
 1.7755488157272339,
 1.769974946975708,
 1.7623789310455322,
 1.7520973682403564,
 1.7384141683578491,
 1.7215535640716553,
 1.6992201805114746,
 1.6700998544692993,
 1.635474443435669,
 1.5951752662658691,
 1.5549689531326294,
 1.5136748552322388,
 1.4732840061187744,
 1.4271153211593628,
 1.3893206119537354,
 1.34617280960083,
 1.308599591255188,
 1.2698791027069092,
 1.2361537218093872,
 1.2006230354309082,
 1.166331171989441,
 1.1322721242904663,
 1.1027084589004517,
 1.0737736225128174,
 1.0481826066970825,
 1.02433180809021,
 1.000127911567688,
 0.9798629879951477,
 0.9611186981201172,
 0.9449626803398132,
 0.9279831051826477,
 0.9136921167373657,
 0.8993396759033203,
 0.8908110857009888,
 0.8788508176803589,
 0.8698298335075378,
 0.8557330965995789,
```

```
0.843579888343811,
0.8326022028923035,
0.8261652588844299,
0.8145310878753662,
0.8079551458358765,
0.7973245978355408,
0.7895845174789429,
0.7816458344459534,
0.7739992141723633,
0.7664980292320251,
0.7600471377372742,
0.7539140582084656,
0.7476783990859985,
0.7436589598655701,
0.7376172542572021,
0.7318533658981323,
0.7258718013763428,
0.7200705409049988,
0.7149202823638916,
0.7089365124702454,
0.7035531401634216,
0.6982213258743286,
0.6929188966751099,
0.688211977481842,
0.6838538646697998,
0.6793445944786072,
0.6749556064605713,
0.6707113981246948,
0.6657694578170776,
0.6614328026771545,
0.6578323245048523,
0.6525154113769531,
0.6495441794395447,
0.6458582282066345,
0.6426011323928833,
0.6385653018951416,
0.634661853313446,
0.6313019394874573,
0.627489447593689,
0.6250017881393433,
0.6211888790130615,
0.6183204650878906,
0.6145350337028503,
0.6116122007369995,
0.6086138486862183,
0.6057754755020142,
0.6028879284858704,
```

```
 0.5996938943862915,
 0.5971559882164001,
 0.5937790274620056,
 0.5913156867027283,
 0.5885327458381653,
 0.5852164030075073,
 0.5832130312919617,
 0.5801935791969299,
 0.5780117511749268,
 0.574122428894043,
 0.5726749897003174,
 0.5692090392112732,
 0.5682054162025452],
'val_accuracy': [0.20000000298023224,
 0.18333333730697632,
 0.18333333730697632,
 0.23333333432674408,
 0.2666666805744171,
 0.28333333134651184,
 0.3499999940395355,
 0.36666667461395264,
 0.4000000059604645,
 0.4000000059604645,
 0.4000000059604645,
 0.4166666567325592,
 0.42500001192092896,
 0.4749999940395355,
 0.4583333432674408,
 0.4583333432674408,
 0.4833333194255829,
 0.5,
 0.5083333253860474,
 0.5083333253860474,
 0.550000011920929,
 0.574999988079071,
 0.625,
 0.625,
 0.6499999761581421,
 0.6416666507720947,
 0.6416666507720947,
 0.6583333611488342,
 0.6666666865348816,
 0.6916666626930237,
 0.7083333134651184,
 0.699999988079071,
 0.6916666626930237,
 0.7250000238418579,
```

0.7083333134651184,
0.7250000238418579,
0.7250000238418579,
0.7083333134651184,
0.7250000238418579,
0.7166666388511658,
0.7166666388511658,
0.7166666388511658,
0.7166666388511658,
0.7083333134651184,
0.7083333134651184,
0.7166666388511658,
0.7083333134651184,
0.7166666388511658,
0.7250000238418579,
0.7166666388511658,
0.7166666388511658,
0.7250000238418579,
0.7250000238418579,
0.7250000238418579,
0.7250000238418579,
0.7250000238418579,
0.7250000238418579,
0.7333333492279053,
0.7333333492279053,
0.7250000238418579,
0.7416666746139526,
0.7416666746139526,
0.75,
0.75,
0.75,
0.7583333253860474,
0.7583333253860474,
0.7666666507720947,
0.7666666507720947,
0.7666666507720947,
0.7666666507720947,
0.7749999761581421,
0.7749999761581421,
0.7749999761581421,
0.7749999761581421,
0.7833333611488342,
0.7833333611488342,
0.7833333611488342,
0.7833333611488342,
0.7666666507720947,
0.7833333611488342,

```
      0.7749999761581421,
      0.7666666507720947,
      0.7666666507720947,
      0.7666666507720947,
      0.7666666507720947,
      0.7666666507720947,
      0.7666666507720947,
      0.7666666507720947,
      0.7666666507720947,
      0.7666666507720947,
      0.7666666507720947,
      0.7666666507720947,
      0.7749999761581421,
      0.7749999761581421,
      0.7749999761581421,
      0.7833333611488342,
      0.7749999761581421,
      0.7833333611488342,
      0.7916666865348816]}
```

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

```python
[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')]
```

Model loss
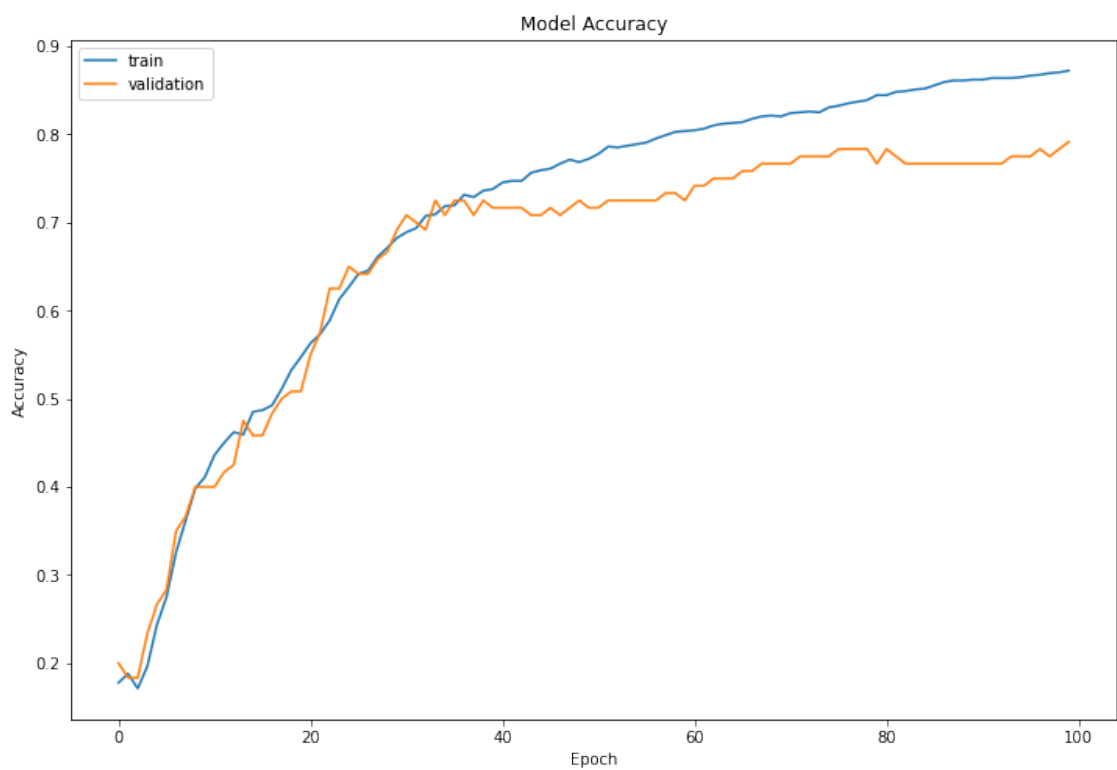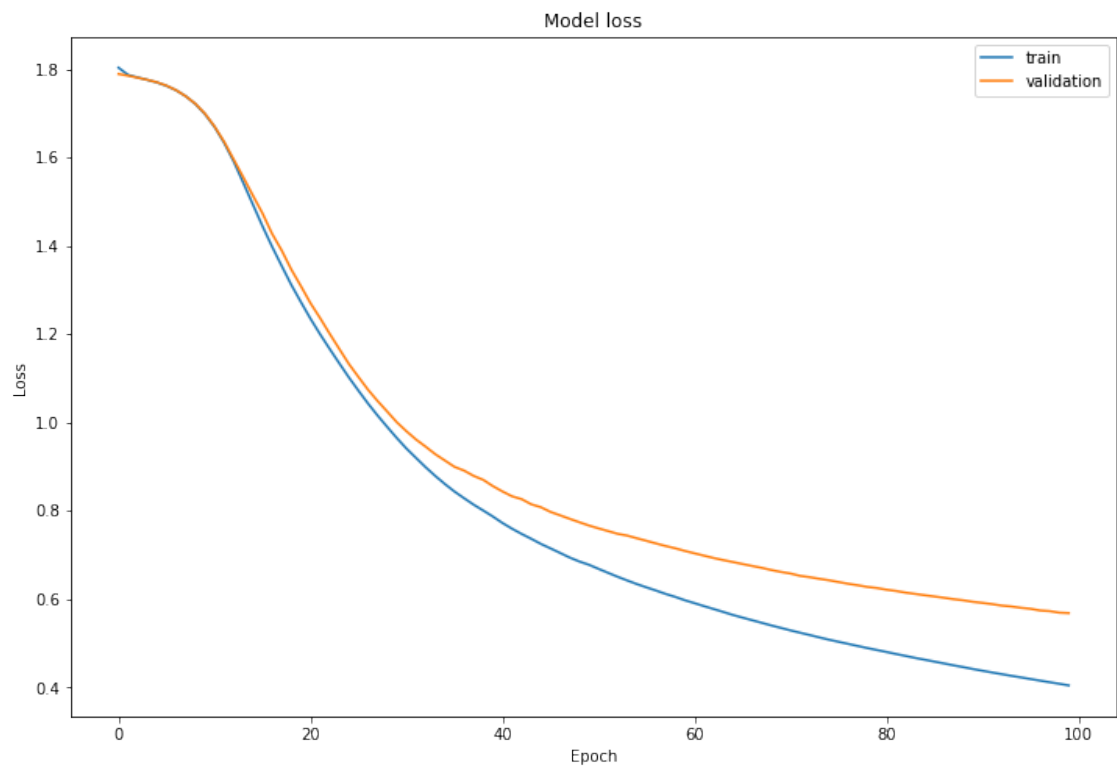
Model Accuracy

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