Artificial Intelligence

3. Convolutional Neural Network (CNN)

Garth Mortensen 2018.10.15

Overview

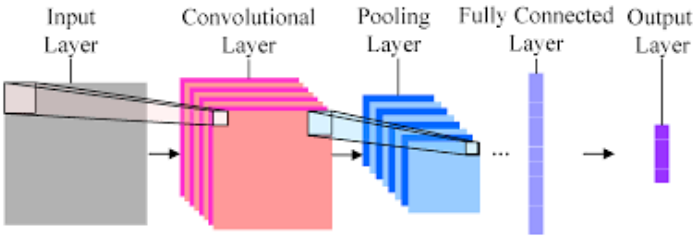
This analysis builds a Convolutional Neural Network (CNN) to classify the Iris flower data set (https://en.wikipedia.org/wiki/Iris_flower_data_set) into three species of Iris.

The *Iris dataset* consists of 150 .jpg images, each 200x200 pixels, 96 dpi, 24 bit depth. The data originates in the 1936 paper *The use of multiple measurements in taxonomic problems* by Ronald Fisher. It's features include length (cm) and width (cm) of the sepals and petals.

Convolutional networks (LeCun 1989), aka *convolutional neural networks* or *CNNs* , are a specialized kind of neural network for processing data that has CNNs a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels.

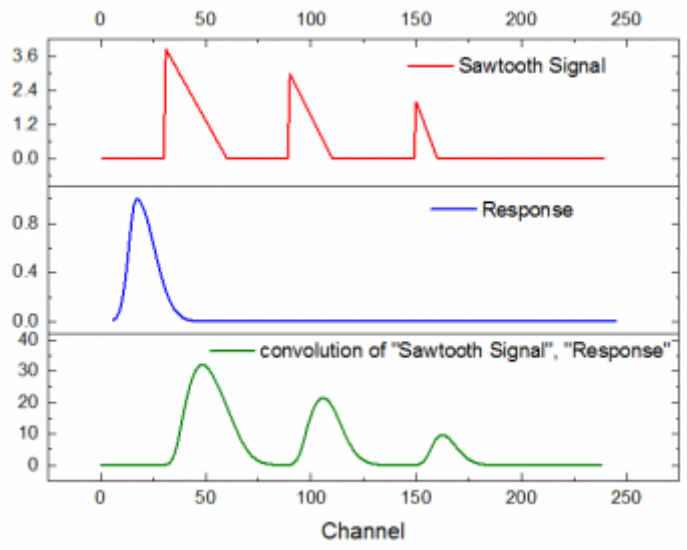
CNNs are used primarily for image data, for self driving cars, image search, video classification, and to a lesser extent, voice recognition and natural language processing. CNNs were inspired by Fukushima's 1980 Neocognitron (https://en.wikipedia.org/wiki/Neocognitron), which was inpired by 1958/1959 experiments by Hubel and Wiesel (https://www.youtube.com/watch?v=4nwpU7GFYe8) on primates on the visual cortex.

According to Geron (359), then came an LeCun et al. in 1998, introducing LeNet-5's CNN architecture, It introduced *convolutional layers* and *pooling layers*.



What is convolution? It is a specialized kind of linear operation. An example of a convolution operation is provided by Goodfellow (322)

> Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output x (t), the position of the spaceship at time t . Both x and t are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.
>
> Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function w (a), where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function providing a smoothed estimate of the position of the spaceship.
>
> ...
>
> In general, convolution is defined for any functions for which the above integral is defined, and may be used for other purposes besides taking weighted averages.



Expressed another way, *a convolution is a small tensor that can be multiplied over a little section of the main image. They're also called filters, because depending on the values in that convolution array, it can pick out specific patterns from the image.*

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. Given the previous simplified definition, we could be adventurous and even call these filter networks. They also include the building blocks of *padding*, or adding 0 weighted borders in order to maintain a kernels dimensions around the images borders, and *stride*, or distance between consequtive receptive fields (filters). Finally, the feature map highlights the areas in an image that are most similar to the filter (Geron, 364).

For a video introduction to convolutions, see here (https://www.youtube.com/watch?v=OVbiVIChkVY)

Sources:

- Deep Learning with Python, Chapter 5
- Hands On Machine Learning with Scikit Learn, Chapter 13
- Deep Learning, Chapter 9 (pg 330)

A Note on Reproducibility

Reproducibility is crucial in science. Using a programmatic approach to loading and manipulating data, modeling and validation can help future researchers (and yourself) reproduce your findings. Don't believe me?

> Brian Nosek of University of Virginia and colleagues sought out to replicate 100 different studies that all were published in 2008.[2] The project pulled these studies from three different journals, Psychological Science, the Journal of Personality and Social Psychology, and the Journal of Experimental Psychology: Learning, Memory, and Cognition, published in 2008 to see if they could get the same results as the initial findings. In their initial publications 97 of these 100 studies claimed to have significant results. To stay as true as they could the group went through extensive measures to remain true to the original studies, to the extent of consulting the original authors. Even with all the extra steps taken to ensure the same conditions of the original 97 studies only 35 of the studies replicated (36.1%), and if they did replicate their effects were smaller than the initial studies effects. The authors emphasized that the findings reflect a problem that affects all of science not just psychology, and that there is room to improve reproducibility in psychology.

> There have been multiple implications of the Reproducibility Project. People all over have started to question the legitimacy of scientific studies that have been published in esteemed journals. Journals typically only publish articles with big effect sizes that reject the null hypothesis. Leading into the huge issue of people re-doing studies that have already found to fail, but not knowing because there is no record of the failed studies, which will lead to more false positives to be published. It is unknown if any of the original study authors committed fraud in publishing their projects, but some of the authors of the original studies are part of the 270 contributors of this project. (source (https://en.wikipedia.org/wiki/Reproducibility_Project)) or (interview (http://www.econtalk.org/brian-nosek-on-the-reproducibility-project/))

Objectives:

1. Download image dataset (Iris_Imgs.7z).
2. Write a program using Python Keras, MatLab, or any programming language of your choice to classify images into 3 classes (setosa, versicolor, virginica) using CNN.
3. Print the configurations of all the layers in your CNN.
4. Print the confusion matrix of your classification result, and what is the accuracy of classification result? The accuracy can be obtained either from training data, test data, or both. Please specify how the accuracy being calculated.
5. Save your code as “a3.py”, “a3.ipynb”, or “a3.m”.

Pre-Modeling

Load packages

We begin by importing some of the required libraries.

- **NumPy** Allows you to efficiently work with data in arrays. It provides the foundation data structures and operations for SciPy. These are arrays (ndarrays) that are efficient to define and manipulate.
- **Pandas** Tools and data structures to organize and analyze data. The key to understanding Pandas for machine learning is understanding the Series and DataFrame data structures.
- **Matplotlib** Allows you to create 2D charts and plots.
  - Call a plotting function with some data (e.g. .plot()).
  - Call many functions to setup the properties of the plot (e.g. labels and colors).
  - Make the plot visible (e.g. .show()).
  - *Not used*
- **SKLearn** has machine learning algorithms for classification, regression, clustering and more. It also has tools for related tasks such as evaluating models, tuning parameters and pre-processing. Scikit-learn is open source and is usable commercially under the BSD license.
  - *Not used*
- **Tensorflow** is a more complex library for distributed numerical computation using data flow graphs. You can train and run very large neural networks efficiently by distributing the computations across potentially thousands of multi-GPU servers. **Tensor is the word for something that is like a matrix, but can have any number of dimensinos.**
- **Keras** is tensorflow's front-end.

Other packages are described as they called on.

In [173]:

```
# Basic math libraries
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Import library
import os
print(os.listdir("../input"))
# Take note that the .7z directory name is automatically converted to lowercase.

import os.path
# Ensure we're reading the directory correctly.
print("os.path identified:")
os.path.exists("../input/iris_imgs/Iris_Imgs/")
```

```
['iris_imgs']
os.path identified:

True
```

Load Data

An extreme shortcut to loading this dataset can be found in this kernel (https://www.kaggle.com/jgrandinetti/classification-using-convnet-keras). ImageDataGenerator uses very few lines of code, as opposed to pages of code in my previous version.

Define the training and test paths, which are the same for this dataset.

In [174]:
```
# script_dir = os.path.dirname(".")
training_set_path = os.path.join('../input/iris_imgs/Iris_Imgs/')
test_set_path = os.path.join('../input/iris_imgs/Iris_Imgs/')
```

Instantiate a datagenerator.

In [175]:
```
from tensorflow.python.keras.preprocessing.image import ImageDataGenerator

data_generator = ImageDataGenerator() # vanilla approach
```

Time to load the data. We define some variables ahead of time for cleaner code further down.

In [176]:
```
batch_size = 10 # impacts processing time
image_size = 200 # image size is 200x200

train_datagen = ImageDataGenerator(rescale = 1. / 150, # rescale is optional
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)

test_datagen = ImageDataGenerator(rescale = 1. / 150, validation_split = 0.3) # rescale is optional
```

Everything worked? Proceed. Now execute the split and label the datasets.

In [177]:
```
training_set  = train_datagen.flow_from_directory(
        '../input/iris_imgs/Iris_Imgs/',
        target_size = (image_size, image_size),
        batch_size  = batch_size,
        subset      = "training")  # keep data in same order as labels)

test_set  = test_datagen.flow_from_directory(
        '../input/iris_imgs/Iris_Imgs/',
        target_size = (image_size, image_size),
        batch_size  = batch_size,
        subset      = "validation")  # keep data in same order as labels)
```

```
Found 150 images belonging to 3 classes.
Found 45 images belonging to 3 classes.
```

Potential delete cell

Later on, we will be interested in using labels while creating the confusion matrix. [This (https://stackoverflow.com/questions/48373685/keras-imagedatagenerator-how-to-get-all-labels-from-data)](https://stackoverflow.com/questions/48373685/keras-imagedatagenerator-how-to-get-all-labels-from-data) useful SO post describes how to obtain them.

In [207]:
```
import math

number_of_examples = len(test_set.filenames)
number_of_generator_calls = math.ceil(number_of_examples / (1.0 * batch_size))

test_labels = []

for i in range(0,int(number_of_generator_calls)):
    test_labels.extend(np.array(test_set[i][1]))
```

```
[array([0., 0., 1.], dtype=float32), array([1., 0., 0.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 1., 0.], dtype=float32), array([0., 1., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([0., 1., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([0., 1., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([0., 0., 1.], dtype=float32), array([1., 0., 0.], dtype=float32), array([1., 0., 0.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 1., 0.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 1., 0.], dtype=float32), array([1., 0., 0.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([0., 1., 0.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([0., 1., 0.], dtype=float32), array([0., 1., 0.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([0., 1., 0.], dtype=float32), array([0., 1., 0.], dtype=float32), array([1., 0., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([0., 1., 0.], dtype=float32), array([0., 1., 0.], dtype=float32), array([0., 0., 1.], dtype=float32), array([0., 0., 1.], dtype=float32)]
```

The data is split randomly during our model fitting process. As such, we don't need sklearn data split, and can move directly onto building the model.

Build the CNN Model

> Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps) thanks to the convolutional layers (see Figure 13-9). At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities). (Geron 371)
>
> *A common mistake is to use convolution kernels that are too large. You can often get the same effect as a 9 × 9 kernel by stacking two 3 × 3 kernels on top of each other, for a lot less compute.*
> ...
> Due to CNN architecture, competitive models' top-5 error rate for image classification fell from over 26% to barely over 3% in just five years.

Import CNN libraries

In [178]:
```
# From Notes
# Simple CNN
# import tensorflow as tf
# importing tensorflow seems to be automatic

# Keras libraries
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
```

Model Configuration

In [179]:
```
model = Sequential()

# Conv2D layer (instead of dense) w/ 32 feature maps, w/ 5×5 kernel size.
# 200x200 pixels,
# relu for now, but end with softmax since 3 classes
model.add(Conv2D(32, (5, 5), input_shape = (200, 200, 3), activation = 'relu'))

# MaxPooling2D layer w/ a pool size of 2×2. Max finds max value from each feature map's ReLu output [0, in
model.add(MaxPooling2D(pool_size = (3, 3)))

# Dropout layer randomly excludes 20% of neurons
# Dropout helps decrease amount of overfitting
model.add(Dropout(0.2))

# Flatten layer converts 2D matrix (feature maps) to 1-D vector for the next dense/fully connected (FC) la
model.add(Flatten())

# Dense layer w/ 128 neurons and ReLU
model.add(Dense(128, activation = 'relu'))

# Output (Dense) layer w/ 3 neurons for 3 classes & a softmax activation
model.add(Dense(3, activation = 'softmax'))
```

Let's review the configurations of all the layers in this CNN.

In [180]:
```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)            (None, 196, 196, 32)      2432
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 65, 65, 32)        0
_________________________________________________________________
dropout_8 (Dropout)          (None, 65, 65, 32)        0
_________________________________________________________________
flatten_8 (Flatten)          (None, 135200)            0
_________________________________________________________________
dense_15 (Dense)             (None, 128)               17305728
_________________________________________________________________
dense_16 (Dense)             (None, 3)                 387
=================================================================
Total params: 17,308,547
Trainable params: 17,308,547
Non-trainable params: 0
_________________________________________________________________
```

The number of parameters for pooling, dropout and flatten are zero. The parameter count then blows up after the first flatten.

Compile Model

In [183]:
```
model.compile(loss = keras.losses.categorical_crossentropy,
              optimizer = keras.optimizers.SGD(lr = 0.01),
              metrics = ['accuracy'])
```

Fit Model

In [249]:
```
count = sum([len(files) for r, d, files in os.walk("../input/iris_imgs/Iris_Imgs/")]) # Not so sure about

epochsCount = 100
validation_stepsCount = 100

# Trains the model on data generated batch-by-batch by a Python generator
# The generator is run in parallel to the model, for efficiency.
# For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training
ConvModel = model.fit_generator(
        training_set,
        steps_per_epoch = int(count / batch_size) + 1, # Not so sure about this
        epochs = epochsCount, # Make sure GPU is enabled
        validation_data = test_set,
        validation_steps = validation_stepsCount)
```

```
Epoch 1/100
16/16 [==============================] - 4s 246ms/step - loss: 0.2022 - acc: 0.8937 - val_loss: 0.2235 - val_acc: 0.9111
Epoch 2/100
16/16 [==============================] - 3s 182ms/step - loss: 0.2709 - acc: 0.8625 - val_loss: 0.2082 - val_acc: 0.8889
Epoch 3/100
16/16 [==============================] - 3s 177ms/step - loss: 0.2516 - acc: 0.8812 - val_loss: 0.2114 - val_acc: 0.9111
Epoch 4/100
16/16 [==============================] - 3s 178ms/step - loss: 0.2746 - acc: 0.8750 - val_loss: 0.2564 - val_acc: 0.9111
Epoch 5/100
16/16 [==============================] - 3s 181ms/step - loss: 0.2280 - acc: 0.9000 - val_loss: 0.1922 - val_acc: 0.8889
Epoch 6/100
16/16 [==============================] - 3s 181ms/step - loss: 0.2388 - acc: 0.8875 - val_loss: 0.2006 - val_acc: 0.8889
Epoch 7/100
16/16 [==============================] - 3s 182ms/step - loss: 0.2357 - acc: 0.8875 - val_loss: 0.2160 - val_acc: 0.9111
Epoch 8/100
16/16 [==============================] - 3s 181ms/step - loss: 0.2566 - acc: 0.8750 - val_loss: 0.3125 - val_acc: 0.8889
Epoch 9/100
16/16 [==============================] - 3s 182ms/step - loss: 0.2293 - acc: 0.8937 - val_loss: 0.1914 - val_acc: 0.9333
Epoch 10/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2613 - acc: 0.8750 - val_loss: 0.2355 - val_acc: 0.8667
Epoch 11/100
16/16 [==============================] - 3s 178ms/step - loss: 0.2437 - acc: 0.8875 - val_loss: 0.2025 - val_acc: 0.8667
Epoch 12/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2292 - acc: 0.9000 - val_loss: 0.1599 - val_acc: 0.9333
Epoch 13/100
16/16 [==============================] - 3s 181ms/step - loss: 0.2132 - acc: 0.8812 - val_loss: 0.2129 - val_acc: 0.8667
Epoch 14/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2716 - acc: 0.8937 - val_loss: 0.1601 - val_acc: 0.9556
Epoch 15/100
16/16 [==============================] - 3s 179ms/step - loss: 0.2051 - acc: 0.9250 - val_loss: 0.1378 - val_acc: 0.9778
Epoch 16/100
16/16 [==============================] - 3s 177ms/step - loss: 0.2320 - acc: 0.8938 - val_loss: 0.1568 - val_acc: 0.9333
Epoch 17/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1591 - acc: 0.9437 - val_loss: 0.1036 - val_acc: 0.9778
Epoch 18/100
16/16 [==============================] - 3s 181ms/step - loss: 0.1890 - acc: 0.8937 - val_loss: 0.2108 - val_acc: 0.9333
Epoch 19/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2026 - acc: 0.9062 - val_loss: 0.1278 - val_acc: 0.9333
Epoch 20/100
16/16 [==============================] - 3s 179ms/step - loss: 0.2970 - acc: 0.8562 - val_loss: 0.2162 - val_acc: 0.8667
Epoch 21/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2183 - acc: 0.9125 - val_loss: 0.1513 - val_acc: 0.9111
Epoch 22/100
16/16 [==============================] - 3s 177ms/step - loss: 0.2147 - acc: 0.8875 - val_loss: 0.1463 - val_acc: 0.9111
Epoch 23/100
16/16 [==============================] - 3s 182ms/step - loss: 0.2053 - acc: 0.8937 - val_loss: 0.1804 - val_acc: 0.8889
Epoch 24/100
16/16 [==============================] - 3s 178ms/step - loss: 0.2502 - acc: 0.8937 - val_loss: 0.1596 - val_acc: 0.9111
Epoch 25/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2580 - acc: 0.8812 - val_loss: 0.1293 - val_acc: 0.9778
Epoch 26/100
16/16 [==============================] - 3s 178ms/step - loss: 0.2356 - acc: 0.8750 - val_loss: 0.2827 - val_acc: 0.8222
Epoch 27/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1893 - acc: 0.9000 - val_loss: 0.1634 - val_acc: 0.8889
Epoch 28/100
16/16 [==============================] - 3s 179ms/step - loss: 0.2375 - acc: 0.8750 - val_loss: 0.1856 - val_acc: 0.9333
Epoch 29/100
16/16 [==============================] - 3s 182ms/step - loss: 0.2406 - acc: 0.8875 - val_loss: 0.1796 - val_acc: 0.9333
Epoch 30/100
```

```
16/16 [==============================] - 3s 179ms/step - loss: 0.2238 - acc: 0.8750 - val_loss: 0.1558 - val_acc: 0.9556
Epoch 31/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2016 - acc: 0.9000 - val_loss: 0.2085 - val_acc: 0.9333
Epoch 32/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1821 - acc: 0.8937 - val_loss: 0.1469 - val_acc: 0.9333
Epoch 33/100
16/16 [==============================] - 3s 177ms/step - loss: 0.1924 - acc: 0.9250 - val_loss: 0.1815 - val_acc: 0.9111
Epoch 34/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2658 - acc: 0.8750 - val_loss: 0.1671 - val_acc: 0.9111
Epoch 35/100
16/16 [==============================] - 3s 182ms/step - loss: 0.1859 - acc: 0.9125 - val_loss: 0.1316 - val_acc: 0.9556
Epoch 36/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1853 - acc: 0.9062 - val_loss: 0.1609 - val_acc: 0.8667
Epoch 37/100
16/16 [==============================] - 3s 182ms/step - loss: 0.2055 - acc: 0.8938 - val_loss: 0.1673 - val_acc: 0.8889
Epoch 38/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1986 - acc: 0.9125 - val_loss: 0.1144 - val_acc: 0.9778
Epoch 39/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2612 - acc: 0.8687 - val_loss: 0.1507 - val_acc: 0.9111
Epoch 40/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1979 - acc: 0.9000 - val_loss: 0.1299 - val_acc: 0.9556
Epoch 41/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1812 - acc: 0.9375 - val_loss: 0.1031 - val_acc: 0.9556
Epoch 42/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1671 - acc: 0.9250 - val_loss: 0.1321 - val_acc: 0.9333
Epoch 43/100
16/16 [==============================] - 3s 181ms/step - loss: 0.2351 - acc: 0.9000 - val_loss: 0.1132 - val_acc: 0.9778
Epoch 44/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2043 - acc: 0.9000 - val_loss: 0.1306 - val_acc: 0.9556
Epoch 45/100
16/16 [==============================] - 3s 181ms/step - loss: 0.2172 - acc: 0.8812 - val_loss: 0.2149 - val_acc: 0.9333
Epoch 46/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1418 - acc: 0.9437 - val_loss: 0.1207 - val_acc: 0.9778
Epoch 47/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1913 - acc: 0.9250 - val_loss: 0.1272 - val_acc: 0.9556
Epoch 48/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1636 - acc: 0.9375 - val_loss: 0.0888 - val_acc: 0.9778
Epoch 49/100
16/16 [==============================] - 3s 179ms/step - loss: 0.2653 - acc: 0.8687 - val_loss: 0.1751 - val_acc: 0.9333
Epoch 50/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1460 - acc: 0.9375 - val_loss: 0.0819 - val_acc: 0.9778
Epoch 51/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1837 - acc: 0.9000 - val_loss: 0.1223 - val_acc: 0.9778
Epoch 52/100
16/16 [==============================] - 3s 178ms/step - loss: 0.2536 - acc: 0.9000 - val_loss: 0.1292 - val_acc: 0.9556
Epoch 53/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1790 - acc: 0.9250 - val_loss: 0.1156 - val_acc: 0.9778
Epoch 54/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1671 - acc: 0.9062 - val_loss: 0.2504 - val_acc: 0.8667
Epoch 55/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1945 - acc: 0.9312 - val_loss: 0.1443 - val_acc: 0.9333
Epoch 56/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1985 - acc: 0.9000 - val_loss: 0.1088 - val_acc: 0.9778
Epoch 57/100
16/16 [==============================] - 3s 181ms/step - loss: 0.1590 - acc: 0.9250 - val_loss: 0.1774 - val_acc: 0.8889
Epoch 58/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1505 - acc: 0.9250 - val_loss: 0.0847 - val_acc: 0.9778
Epoch 59/100
16/16 [==============================] - 3s 185ms/step - loss: 0.2154 - acc: 0.8875 - val_loss: 0.1625 - val_acc: 0.8889
Epoch 60/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2008 - acc: 0.8875 - val_loss: 0.1726 - val_acc: 0.9333
Epoch 61/100
16/16 [==============================] - 3s 181ms/step - loss: 0.1241 - acc: 0.9562 - val_loss: 0.0825 - val_acc: 0.9778
Epoch 62/100
16/16 [==============================] - 3s 179ms/step - loss: 0.2205 - acc: 0.8875 - val_loss: 0.1573 - val_acc: 0.9111
Epoch 63/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1565 - acc: 0.9312 - val_loss: 0.1283 - val_acc: 0.9333
Epoch 64/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1400 - acc: 0.9437 - val_loss: 0.0866 - val_acc: 0.9778
Epoch 65/100
16/16 [==============================] - 3s 180ms/step - loss: 0.2006 - acc: 0.9062 - val_loss: 0.1301 - val_acc: 0.9333
Epoch 66/100
16/16 [==============================] - 3s 181ms/step - loss: 0.1778 - acc: 0.9062 - val_loss: 0.1172 - val_acc: 0.9333
Epoch 67/100
16/16 [==============================] - 3s 184ms/step - loss: 0.2164 - acc: 0.8875 - val_loss: 0.1310 - val_acc: 0.9333
Epoch 68/100
16/16 [==============================] - 3s 182ms/step - loss: 0.2159 - acc: 0.9187 - val_loss: 0.1058 - val_acc: 0.9556
Epoch 69/100
16/16 [==============================] - 3s 183ms/step - loss: 0.1282 - acc: 0.9375 - val_loss: 0.1306 - val_acc: 0.9333
Epoch 70/100
16/16 [==============================] - 3s 183ms/step - loss: 0.1181 - acc: 0.9562 - val_loss: 0.1488 - val_acc: 0.9111
Epoch 71/100
```

```
16/16 [==============================] - 3s 183ms/step - loss: 0.2252 - acc: 0.9125 - val_loss: 0.1638 - val_acc: 0.9111
Epoch 72/100
16/16 [==============================] - 3s 182ms/step - loss: 0.1835 - acc: 0.9125 - val_loss: 0.1208 - val_acc: 0.9778
Epoch 73/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1580 - acc: 0.9250 - val_loss: 0.1162 - val_acc: 0.9333
Epoch 74/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1989 - acc: 0.8875 - val_loss: 0.1307 - val_acc: 0.9333
Epoch 75/100
16/16 [==============================] - 3s 178ms/step - loss: 0.2179 - acc: 0.8937 - val_loss: 0.1441 - val_acc: 0.9333
Epoch 76/100
16/16 [==============================] - 3s 182ms/step - loss: 0.1896 - acc: 0.9125 - val_loss: 0.0898 - val_acc: 0.9778
Epoch 77/100
16/16 [==============================] - 3s 178ms/step - loss: 0.2044 - acc: 0.9312 - val_loss: 0.1269 - val_acc: 0.9333
Epoch 78/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1968 - acc: 0.9000 - val_loss: 0.1820 - val_acc: 0.9111
Epoch 79/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1667 - acc: 0.9000 - val_loss: 0.0881 - val_acc: 0.9778
Epoch 80/100
16/16 [==============================] - 3s 181ms/step - loss: 0.2083 - acc: 0.8812 - val_loss: 0.1459 - val_acc: 0.9333
Epoch 81/100
16/16 [==============================] - 3s 182ms/step - loss: 0.1361 - acc: 0.9375 - val_loss: 0.1864 - val_acc: 0.8889
Epoch 82/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1515 - acc: 0.9250 - val_loss: 0.1346 - val_acc: 0.9556
Epoch 83/100
16/16 [==============================] - 3s 181ms/step - loss: 0.2496 - acc: 0.9062 - val_loss: 0.1185 - val_acc: 0.9556
Epoch 84/100
16/16 [==============================] - 3s 177ms/step - loss: 0.1617 - acc: 0.9125 - val_loss: 0.1687 - val_acc: 0.8889
Epoch 85/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1692 - acc: 0.9062 - val_loss: 0.1270 - val_acc: 0.9778
Epoch 86/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1317 - acc: 0.9375 - val_loss: 0.0930 - val_acc: 0.9778
Epoch 87/100
16/16 [==============================] - 3s 178ms/step - loss: 0.1544 - acc: 0.9312 - val_loss: 0.1276 - val_acc: 0.9333
Epoch 88/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1365 - acc: 0.9312 - val_loss: 0.1832 - val_acc: 0.8667
Epoch 89/100
16/16 [==============================] - 3s 176ms/step - loss: 0.1970 - acc: 0.9000 - val_loss: 0.1484 - val_acc: 0.9556
Epoch 90/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1960 - acc: 0.9250 - val_loss: 0.1261 - val_acc: 0.9333
Epoch 91/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1740 - acc: 0.9062 - val_loss: 0.1450 - val_acc: 0.9333
Epoch 92/100
16/16 [==============================] - 3s 176ms/step - loss: 0.1989 - acc: 0.9187 - val_loss: 0.2483 - val_acc: 0.9111
Epoch 93/100
16/16 [==============================] - 3s 179ms/step - loss: 0.1864 - acc: 0.9125 - val_loss: 0.1000 - val_acc: 0.9778
Epoch 94/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1547 - acc: 0.9250 - val_loss: 0.0807 - val_acc: 0.9778
Epoch 95/100
16/16 [==============================] - 3s 177ms/step - loss: 0.1696 - acc: 0.9000 - val_loss: 0.1607 - val_acc: 0.9333
Epoch 96/100
16/16 [==============================] - 3s 186ms/step - loss: 0.1942 - acc: 0.8937 - val_loss: 0.1285 - val_acc: 0.9111
Epoch 97/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1214 - acc: 0.9312 - val_loss: 0.0834 - val_acc: 0.9778
Epoch 98/100
16/16 [==============================] - 3s 180ms/step - loss: 0.1300 - acc: 0.9437 - val_loss: 0.1726 - val_acc: 0.9111
Epoch 99/100
16/16 [==============================] - 3s 181ms/step - loss: 0.1519 - acc: 0.9125 - val_loss: 0.1679 - val_acc: 0.9111
Epoch 100/100
16/16 [==============================] - 3s 181ms/step - loss: 0.1474 - acc: 0.9187 - val_loss: 0.0753 - val_acc: 0.9778
```

Let's also save this for future use.

In [250]:
```
model.save('Iris_CNN')

# We load the model using "load_weights"
# classifier.load_weights('flower_recogintion.h5')
```

Evaluate Model

In [251]:
```
acc = ConvModel.history['acc']
val_acc = ConvModel.history['val_acc']
loss = ConvModel.history['loss']
val_loss = ConvModel.history['val_loss']
```

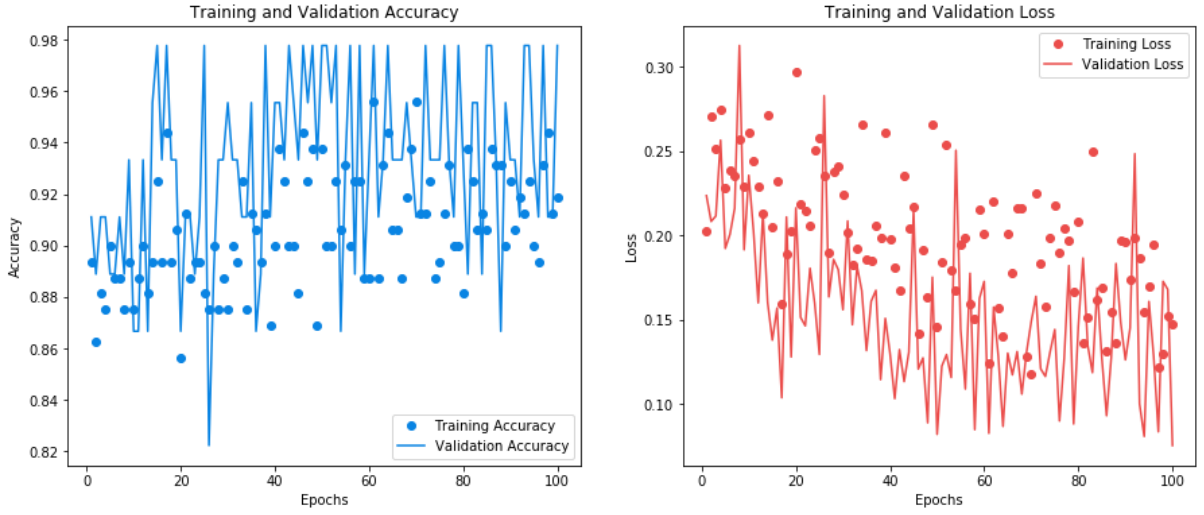
In [252]:
```
import matplotlib.pyplot as plt

epochs = range(1, len(acc) + 1)

plt.figure(figsize = (15, 6));
plt.subplot(1,2,1)
plt.plot(epochs, acc, color = '#0984e3', marker = 'o', linestyle = 'none', label = 'Training Accuracy')
plt.plot(epochs, val_acc, color = '#0984e3', label = 'Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend(loc = 'best')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1,2,2)
plt.plot(epochs, loss, color = '#eb4d4b', marker = 'o',linestyle = 'none',label = 'Training Loss')
plt.plot(epochs, val_loss, color = '#eb4d4b', label = 'Validation Loss')
plt.title('Training and Validation Loss')
plt.legend(loc = 'best')
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.show()
```



What is the final accuracy of this model? It consists of two parts, training and validation.

In [253]:
```
print("The final training accuracy of this model is", acc[-1])
print("The final validation accuracy of this model is", val_acc[-1])
```

```
The final training accuracy of this model is 0.9187499918043613
The final validation accuracy of this model is 0.977777773141861
```

Let's try to do a confusion matrix as well. [This (https://datascience.stackexchange.com/questions/13894/how-to-get-predictions-with-predict-generator-on-streaming-test-data-in-keras)](https://datascience.stackexchange.com/questions/13894/how-to-get-predictions-with-predict-generator-on-streaming-test-data-in-keras) SO post might work. For the confusion matrix, you have to use sklearn package.

1. Make predictions for the test data
2. Compute the confusion matrix based on the label predictions

In [254]:
```
from sklearn.metrics import confusion_matrix

# I think the second value is related to batch, validation or epoch amount.
probabilities = model.predict_generator(test_set, 3)
print(len(probabilities))

y_pred = np.argmax(probabilities, axis=1)
print(y_pred)

confusion_matrix(y_true, y_pred)
```

```
30
[0 1 0 1 0 1 2 2 0 1 0 1 2 2 2 0 2 0 1 0 1 1 0 2 0 2 2 0 1 0]

 array([[4, 4, 2],
        [4, 2, 4],
        [4, 3, 3]])
```

The confusion matrix is confused, as you would have predicted.

The End

In [ ]: