kaggle Search kaggle Competitions Datasets Kernels Discussion Learn

Garth

Transfer Learning: Classify Iris

forked from Convolutional Neural Network: Classify Iris by Garth (+0/−0)

last run 4 hours ago · IPython Notebook HTML · 45 views

using data from multiple data sources · Public Make Private

0 voters

Notebook Code Data (2) Output Comments (0) Log Versions (49) Forks Options Fork Notebook Ed

Tags neural networks ✖ image data ✖ image processing ✖ artificial intelligence ✖ multiple data sources Add Tag

Notebook

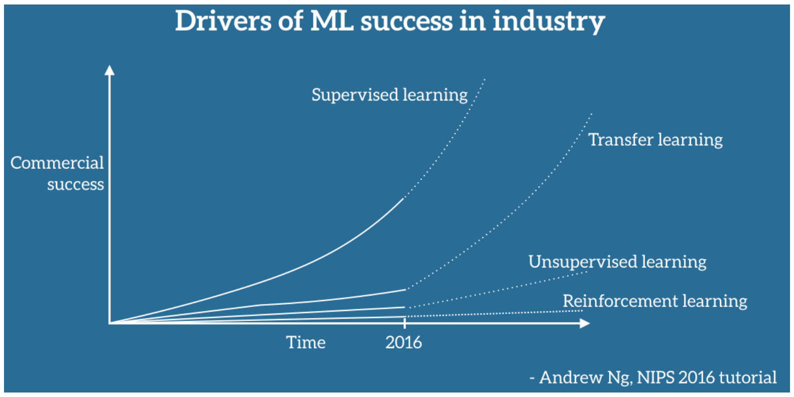
Artificial Intelligence

Transfer Learning: Classify Iris

10/29/2018
Garth Mortensen

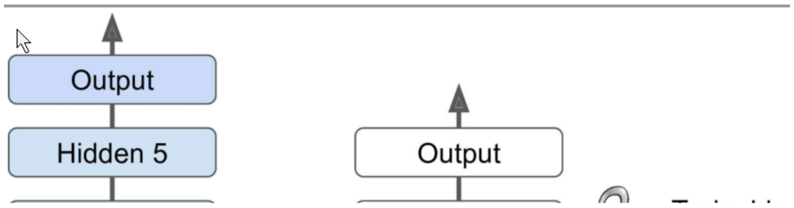
Overview

Transfer Learning is the method of using a pre-trained model built to solve Task A in order to solve for Task B. According to Andrew Ng, transfer learning has strong potential to drive Artificial Intelligence in the future.



What makes it so valuable? Importing a pre-trained model into your project can save substantial training time, processing power, memory and also boost accuracy. Equally important, it allows you to train a model with far less (expensively labeled) data as well. Imagine you only have 20 images. With such a small number, you can't hope to build a highly predictive model that will work on test data. Transfer learning allows us to quickly build accurate models, with relatively little data. But it's not always needed. If you already have a large dataset and don't mind dedicating the time to train a new model, no problem. The architecture might not be top-notch, but you can still get the job done.

In the same way that pretrained word vectors are popularly used for natural language processing, supervised pretraining is popular for convolutional neural networks (CNNs) trained on the ImageNet dataset. However, In applying a model pretrained on dataset D1 to dataset D2, we assume the variations in the model's original training dataset (D1) are similar to those in the new dataset (D2).



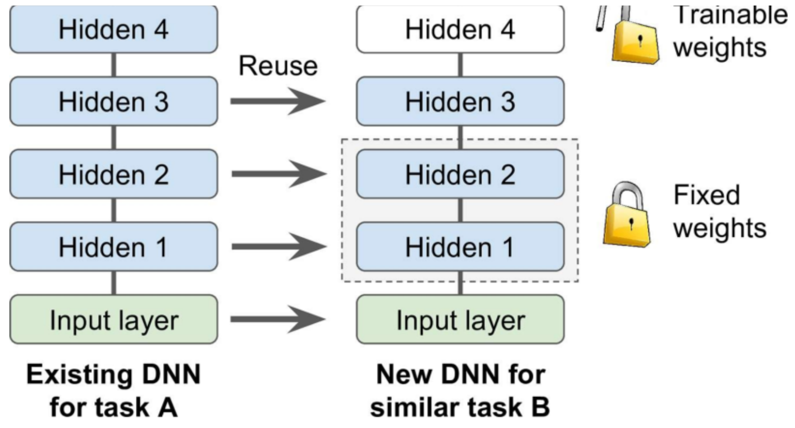


Figure 11-4. Reusing pretrained layers

Goodfellow (p 538) explains how this works:

> For example, we may learn about one set of visual categories, such as cats and dogs, in the first setting, then learn about a different set of visual categories, such as ants and wasps, in the second setting. If there is significantly more data in the first setting (sampled from P1), then that may help to learn representations that are useful to quickly generalize from only very few examples drawn from P2 . Many visual categories share low-level notions of edges and visual shapes, the effects of geometric changes, changes in lighting, etc.

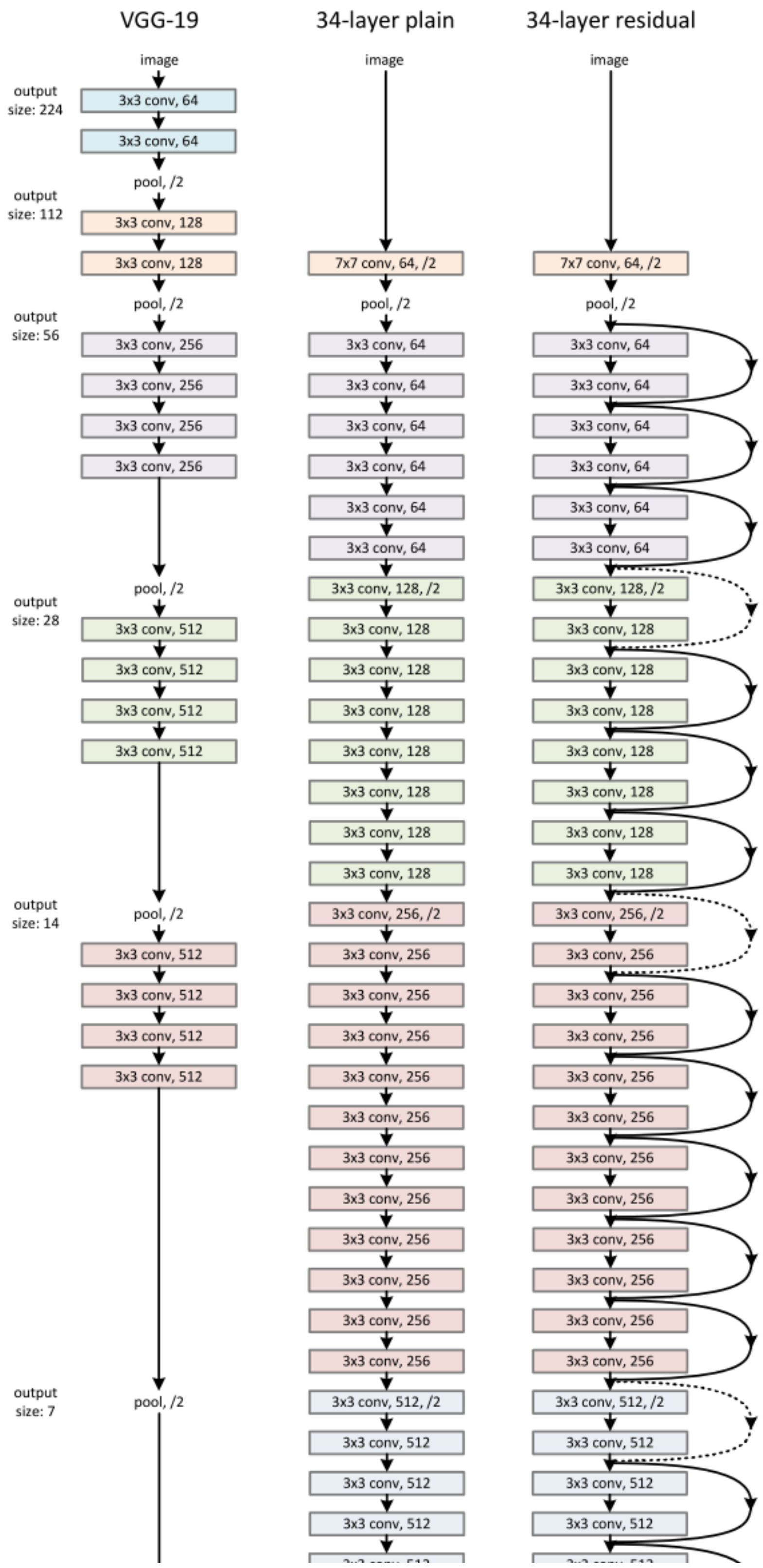
Aurélien Ageron (p 289) makes the strong case that "It is generally not a good idea to train a very large Deep Neural Network (DNN) from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle, then just reuse the lower layers of this network." According to Lisa Torrey and Jude Shavlik (http://pages.cs.wisc.edu/~shavlik/abstracts/torrey.handbook09.abstrac the three ways transfer learning might improve learnings are that it immediately increases your performance, boosts your learning rate (slope), and achieves a higher performance level, as shown in this second chart.

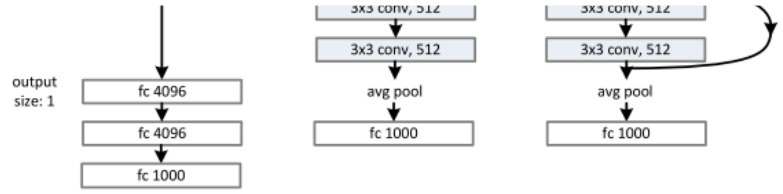
Practitioners publish the parameters of their trained networks to make their work publically available. Keras has 10 pretrained models you can import listed here (https://keras.io/applications/), including:

1. Xception
2. VGG16
3. VGG19
4. **ResNet50**
5. InceptionV3
6. InceptionResNetV2
7. MobileNet
8. DenseNet
9. NASNet

10. MobileNetV2

ResNet50 (https://keras.io/applications/#resnet50) is imported into this notebook. ResNet50 is a pretrained, deep residual network which achieved 3.57% error on ImageNet (http://www.image-net.org/), an image set consisting of handwritten numbers (0-9). You can explore ResNet50's complex architecture here (http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006).





According to Andrew Ng (https://www.youtube.com/watch?v=K0uoBKBQ1gA), DNNs are difficult to train because of vanishing and exploding gradient descents. Residual Networks (resnets) use skip connections, which allow the activation from one layer to feed into deeper layers. This allows for deeper weights to overcome the vanishing or exploding gradient descent problems.

In this notebook, our mission is to use one of these pre-trained models to classify the Iris dataset. 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. However, this Iris dataset has already been converted from .csv to .jpg, making it appropriate for a CNN model.

The set of Iris species .jpg (visualized later in the notebook) is very different from the original ImageNet image set which ResNet50 was trained on, making it an arguably inappropriate application of the pretrained network. *We're assume the variations in ResNet's handwritten numbers dataset (D1) are similar to those in the Iris species dataset (D2)*. Transfer learning works best when the use case has similar visual patterns to the data used in the pre-trained model. However, this notebook will put this idea to the test. Our model will use the following architecture:

In → ResNet50 → Batch Normalization → Dense → Out

ResNet50 comes out of the blackbox with with 2,358,7712 pre-trained, or *frozen* weights. The batch normalization and dense layers add 10,243 parameters, for 0.04% of the total. With such an minute share of total parameters being trained, it will be interesting to see what level of accuracy can be achieved.

Constructing the model will be only half of the work. Before we get elbows deep in parameters, we must first prep the directory and data for processing. As such, the notebook starts with a segment on creating a directory structure and reallocating files into it. The folder restructuing is methodical and clear. The project's workflow can be summarized as follows:

1. Folder Restructuring
2. Data shuffling and splitting
3. Model Building
4. Model Evaluation

Sources:

- Deep Learning with Python, François Chollet
- Deep Learning, Ian Goodfellow, Yoshua Bengio, Aaron Courville
- Hands On Machine Learning with Scikit Learn, Aurélien Géron

Pre-Modeling

Import packages

This notebook requires many libraries. Here's a short introduction on some of the main ones:

- **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. Iirc Keras can also use Theano as an alternative backend to Tensorflow. **Tensor is the word for something that is like a matrix, but can have any number of dimensions.**
- **Keras** is tensorflow's front-end.
- **Other packages** are imported and described as needed.

Import Data

- **Iris** is a dataset uploaded specifically for this and other notebooks.
- **ResNet** is a CNN imported for this notebook.

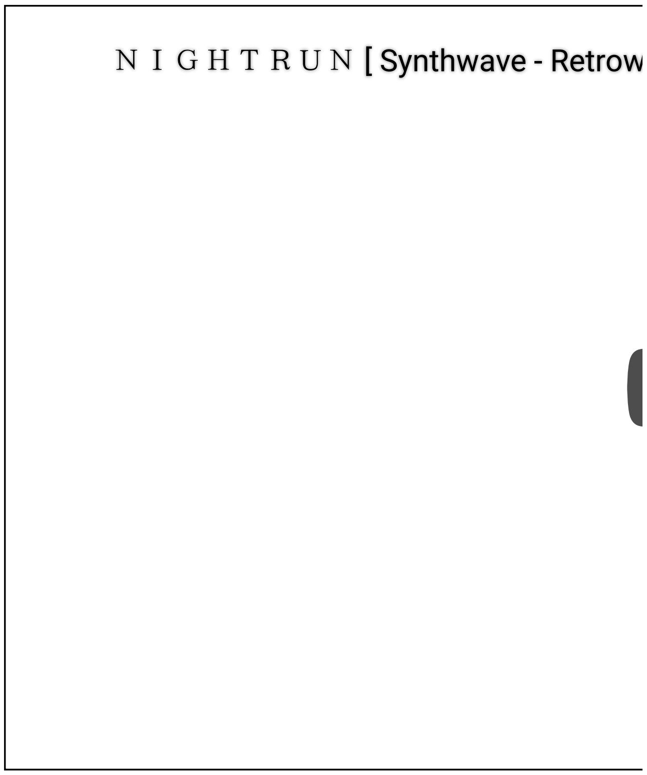
Note: When adding multiple datasets to kaggle, the directory naming convention ***will*** *change. Use os.listdir to figure out how to navigate through the folders.*

Import Sound

In [1]:

```
from IPython.display import YouTubeVideo
YouTubeVideo('arzw0tFK5O4', width = 800, height = 450
)
```

Out[1]:



In [2]:

```
# 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("root paths:", os.listdir("../input"), "\n")
# Take note that the .7z directory name is automatical
ly converted to lowercase,
# and special characters are removed.

import os.path
# Ensure we're reading the directory correctly.
print("os.paths identified:")
print("Resnet model:", os.path.exists("../input/resne
t50/"))
print("Iris dataset:", os.path.exists("../input/iris-
as-images/"))
```

```
root paths: ['resnet50', 'iris-as-image
s']

os.paths identified:
Resnet model: True
Iris dataset: True
```

Why Reorganize the Directory?

When working with many, large images, hardware limitations can become an issue. Though the included dataset is quite small, this notebook serves as an opportunity to build a script which can handle larger datasets. Data_generator, used to load training and test data, is capable of handling large image sets. However, it requires the images be organized in a folder directory such that there are two parent folders train and test, and each one contains segregated class folders. Here, our segregated class folders are setosa, versicolor and virginica. In order to meet this requirement, a new folder structure has to be created, and then images must be copied into them. Moreover, the train-test file count should be split according to your definition of, say, 70-30.

Begin to Reorganize Directory

Point to the various original directory folders

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 my prior versions.

Define the path to our Iris images, and the species subfolders. We'll prefix this set of folders with *orig* to indicate these are the original folders. This is because later, we'll create a directory with images reshuffled into a new directory structure. The folder path will start with root and then branch into species (punny?).

Original directory:

- orig_root_folder/
  - orig_setosa_folder/
  - orig_versicolor_folder/
  - orig_virginica_folder/

In [3]:

```
# main folder
orig_root_folder        = os.path.join('../input/iris-
as-images/iris_imgs/Iris_Imgs/')

print("os.paths identified:")
orig_setosa_folder      = os.path.join(orig_root_folde
r + 'setosa/')
print("setosa:", os.path.exists(orig_setosa_folder))

orig_versicolor_folder = os.path.join(orig_root_folde
r + 'versicolor/')
print("versicolor:", os.path.exists(orig_versicolor_f
older))

orig_virginica_folder  = os.path.join(orig_root_folde
r + 'virginica/')
print("virginica:", os.path.exists(orig_virginica_fol
der))
```

```
os.paths identified:
setosa: True
versicolor: True
virginica: True
```

Create New Directory

Create root folder, and train and test subfolders

Create (https://www.kaggle.com/c/dog-breed-identification/discussion/48908) directories to place train and test folders. The new parent directory structure we'll create:

- new_root_folder/
  - new_train_folder/
  - new_test_folder/

In [4]:

```
import shutil
import shutil

# Create root directory to hold the training and testi
ng child folders
new_root_folder = "Iris_Imgs/"
if not os.path.exists(new_root_folder):
    os.mkdir(new_root_folder)
    print("new root directory created")
else:
    print("new root directory already created")
print("new root directory identified?", os.path.exist
s("Iris_Imgs/"), "\n")

# Create training directory
new_train_folder = "train/"
if not os.path.exists(new_root_folder + new_train_fol
der):
    os.mkdir(new_root_folder + new_train_folder)
    print("new train directory created")
else:
    print("new train directory already created")
print("new train directory identified?", os.path.exis
ts(new_root_folder + new_train_folder), "\n")

# Create testing directory
new_test_folder = "test/"
if not os.path.exists(new_root_folder + new_test_fold
er):
    os.mkdir(new_root_folder + new_test_folder)
    print("new test directory created")
else:
    print("new test directory already created")
print("new test directory identified?", os.path.exist
s(new_root_folder + new_test_folder))
```

```
new root directory created
new root directory identified? True

new train directory created
new train directory identified? True

new test directory created
new test directory identified? True
```

We've created our train and test directories, but still need to add their species subfolders:

- new_root_folder/
  - new_train_folder/

- new_setosa_folder/
- new_versicolor_folder/
- new_virginica_folder/

- new_test_folder/
  - new_setosa_folder/
  - new_versicolor_folder/
  - new_virginica_folder/

Create species subfolders

In [5]:

```
# Create train subfolders

# setosa
new_setosa_folder = "setosa_folder/"
if not os.path.exists(new_root_folder + new_train_fol
der + new_setosa_folder):
    os.mkdir(new_root_folder + new_train_folder + new
_setosa_folder)
    print("new setosa directory created")
else:
    print("new train setosa directory already create
d")
print("new train setosa directory identified?", os.pa
th.exists(new_root_folder + new_train_folder + new_se
tosa_folder), "\n")

# versicolor
new_versicolor_folder = "versicolor_folder/"
if not os.path.exists(new_root_folder + new_train_fol
der + new_versicolor_folder):
    os.mkdir(new_root_folder + new_train_folder + new
_versicolor_folder)
    print("new versicolor directory created")
else:
    print("new versicolor directory already created")
print("new train versicolor directory identified?", o
s.path.exists(new_root_folder + new_train_folder + ne
w_versicolor_folder), "\n")

# virginica
new_virginica_folder = "virginica_folder/"
if not os.path.exists(new_root_folder + new_train_fol
der + new_virginica_folder):
    os.mkdir(new_root_folder + new_train_folder + new
_virginica_folder)
    print("new virginica directory created")
else:
    print("new virginica directory already created")
print("new train virginica directory identified?", os
.path.exists(new_root_folder + new_train_folder + new
_virginica_folder))
```

```
new setosa directory created
new train setosa directory identified? T
rue

new versicolor directory created
new train versicolor directory identifie
d? True

new virginica directory created
new train virginica directory identifie
d? True
```

We have our *train* subfolders, so now we create the *test* subfolders.

In [6]:

```
# Create test subfolders

# setosa
new_setosa_folder = "setosa_folder/"
if not os.path.exists(new_root_folder + new_test_fold
er + new_setosa_folder):
    os.mkdir(new_root_folder + new_test_folder + new_
setosa_folder)
    print("new setosa directory created")
else:
    print("new test setosa directory already created"
)
print("new test setosa directory exists?", os.path.ex
ists(new_root_folder + new_test_folder + new_setosa_f
older), "\n")

# versicolor
new_versicolor_folder = "versicolor_folder/"
if not os.path.exists(new_root_folder + new_test_fold
er + new_versicolor_folder):
    os.mkdir(new_root_folder + new_test_folder + new_
versicolor_folder)
    print("new versicolor directory created")
else:
    print("new versicolor directory already created")
print("new test versicolor directory exists?", os.pat
h.exists(new_root_folder + new_test_folder + new_vers
icolor_folder), "\n")

# virginica
new_virginica_folder = "virginica_folder/"
if not os.path.exists(new_root_folder + new_test_fold
er + new_virginica_folder):
    os.mkdir(new_root_folder + new_test_folder + new_
virginica_folder)
    print("new virginica directory created")
else:
    print("new virginica directory already created")
print("new test virginica directory exists?", os.path
.exists(new_root_folder + new_test_folder + new_virgi
nica_folder))
```

```
new setosa directory created
new test setosa directory exists? True

new versicolor directory created
new test versicolor directory exists? Tr
ue

new virginica directory created
new test virginica directory exists? Tru
e
```

Define new folder paths

Excellent, the new folder structure is complete! The next step is to populate them with images, taken from the original folder structure. As a reminder, we're doing this to conform to the standard folder structure for function ImageDataGenerator.

Original directory:

- orig_root_folder/
  - orig_setosa_folder/
  - orig_versicolor_folder/
  - orig_virginica_folder/

New directory:

- new_root_folder/
  - new_train_folder/
    - new_setosa_folder/
    - new_versicolor_folder/
    - new_virginica_folder/
  - new_test_folder/
    - new_setosa_folder/

- new_versicolor_folder/
- new_virginica_folder/

Before we populate these new folders, we first need to create some pointers to them.

In [7]:

```
new_train_path = os.path.join(new_root_folder + new_t
rain_folder) # used later on in data_generator
new_test_path  = os.path.join(new_root_folder + new_t
est_folder) # used later on in data_generator

orig_setosa_folder_path           = os.path.join(orig
_setosa_folder)
orig_versicolor_folder_path       = os.path.join(orig
_versicolor_folder)
orig_virginica_folder_path        = os.path.join(orig
_virginica_folder)
print("an original path:", orig_setosa_folder_path)

new_train_setosa_folder_path      = os.path.join(new_
root_folder + new_train_folder + new_setosa_folder)
new_train_versicolor_folder_path  = os.path.join(new_
root_folder + new_train_folder + new_versicolor_folde
r)
new_train_virginica_folder_path   = os.path.join(new_
root_folder + new_train_folder + new_virginica_folder
)
print("a new train path:", new_train_versicolor_folde
r_path)

new_test_setosa_folder_path       = os.path.join(new_
root_folder + new_test_folder + new_setosa_folder)
new_test_versicolor_folder_path   = os.path.join(new_
root_folder + new_test_folder + new_versicolor_folder
)
new_test_virginica_folder_path    = os.path.join(new_
root_folder + new_test_folder + new_virginica_folder)
print("a new test path:", new_test_virginica_folder_p
ath)
```

```
an original path: ../input/iris-as-image
s/iris_imgs/Iris_Imgs/setosa/
a new train path: Iris_Imgs/train/versic
olor_folder/
a new test path: Iris_Imgs/test/virginic
a_folder/
```

Copy data from original folders to new folders

With out paths created, the next step is to move the jpgs to their new location.

Immediately below are two historical cells which were used to figure out the process.

In [8]:

```
# # Start with a simple file move
# # We can't use shutil.move because this will copy pa
ste and delete from the original location. Because Kag
gle data directories are read only, this will not wor
k.

# if not os.path.isfile(new_train_setosa_folder_path +
 "Img_1.jpg"):
#     shutil.copy(orig_setosa_folder_path + "Img_1.jp
g", new_train_setosa_folder_path)
#     print("file copied to new directory")
# else:
#     print("Previously copied?", os.path.isfile(new_t
rain_setosa_folder_path + "Img_1.jpg"))

# # Try moving many files
# # [This](https://www.tutorialspoint.com/How-to-copy-
files-from-one-folder-to-another-using-Python) clean l
oop was used.

# files = ["Img_10.jpg", "Img_11.jpg", "Img_12.jpg"]
# moveFrom = orig_setosa_folder_path
# moveTo = new_train_setosa_folder_path

# for f in files:
#     shutil.copy(moveFrom + f, moveTo)
#     print(f, "copied to new directory")
```

Count files

Given a 70-30 train-test split, determine how many files will need to be copied into the new training and testing folders.

In [9]:

```
train_ratio = 0.7
test_ratio  = 0.3
```

```
# setosa
# count total images in folder
path, dirs, files = next(os.walk(orig_setosa_folder_p
ath))
file_count = len(files)
# float for math
file_count = int(file_count) # convert for later use

# determine folder count, according to ratio
setosa_train_file_count = int(file_count * train_rati
o)
setosa_test_file_count  = int(file_count * test_ratio
)
print("train image count:", setosa_train_file_count)
print("test image count:", setosa_test_file_count, "
\n")


# versicolor
# count total images in folder
path, dirs, files = next(os.walk(orig_versicolor_fold
er_path))
file_count = len(files)
# float for math
file_count = int(file_count) #convert for later use

# determine folder count, according to ratio
versicolor_train_file_count = int(file_count * train_
ratio)
versicolor_test_file_count  = int(file_count * test_r
atio)
print("train image count:", versicolor_train_file_cou
nt)
print("test image count:", versicolor_test_file_count
, "\n")


# virginica
# count total images in folder
path, dirs, files = next(os.walk(orig_virginica_folde
r_path))
file_count = len(files)
# float for math
file_count = int(file_count) # convert for later use

# determine folder count, according to ratio
virginica_train_file_count = int(file_count * train_r
atio)
virginica_test_file_count  = int(file_count * test_ra
tio)
```

```
print("train image count:", virginica_train_file_coun
t)
print("test image count:", virginica_test_file_count)
```

```
train image count: 35
test image count: 15

train image count: 35
test image count: 15

train image count: 35
test image count: 15
```

Create list of images from the original species folder paths.

In [10]:

```
setosa_directory = os.listdir(orig_setosa_folder_path
)
print("setosa directory is of data type:", type(setos
a_directory), "\n")

versicolor_directory = os.listdir(orig_versicolor_fol
der_path)
print("versicolor directory is of data type:", type(v
ersicolor_directory), "\n")

virginica_directory = os.listdir(orig_virginica_folde
r_path)
print("virginica directory is of data type:", type(vi
rginica_directory))
```

```
setosa directory is of data type: <class
'list'>

versicolor directory is of data type: <c
lass 'list'>

virginica directory is of data type: <cl
ass 'list'>
```

Randomize lists

Set random seed, then select random n from this new list (https://stackoverflow.com/questions/15511349/select-50-items-from-list-at-random-to-write-to-file/39585770).

In [11]:

```
import random
random.seed(42)

# shuffle the images order
random.shuffle(setosa_directory)
random.shuffle(versicolor_directory)
random.shuffle(virginica_directory)

# select the first 0-n images from shuffled list
setosa_random_train = (setosa_directory[:setosa_train
_file_count]) # prints "n = file_count" random variabl
es
versicolor_random_train = (versicolor_directory[:vers
icolor_train_file_count]) # prints "n = file_count" ra
ndom variables
virginica_random_train = (virginica_directory[:virgin
ica_train_file_count]) # prints "n = file_count" rando
m variables
```

To split the list of directory files, I use a "pythonic" SO (https://stackoverflow.com/questions/4211209/remove-all-the-elements-that-occur-in-one-list-from-another) solution. We take the randomly selected training data and minus it's content from the original directory list, resulting in random_test list.

Including all species together in one code cell would render an disruptively long output, so it is broken into three species. Perform this first on setosa.

In [12]:

```
# setosa
# from all images list, remove the training list, resu
lting in testing list
setosa_random_test = [x for x in setosa_directory if
x not in setosa_random_train]

print("Total images in setosa directory:", len(setosa
_directory))
print(type(setosa_directory))
# print(setosa_directory, "\n")

print("Total images in setosa train list:", setosa_tr
ain_file_count)
print(type(setosa_random_train))
# print(setosa_random_train, "\n")
# print(setosa_random_train, "\n")

print("Total images in setosa test list:", len(setosa
_random_test))
print(type(setosa_random_test))
# print(setosa_random_test)
```

```
Total images in setosa directory: 50
<class 'list'>
Total images in setosa train list: 35
<class 'list'>
Total images in setosa test list: 15
<class 'list'>
```

Now Versicolor.

In [13]:

```
# versicolor
# from all images list, remove the training list, resu
lting in testing list
versicolor_random_test = [x for x in versicolor_direc
tory if x not in versicolor_random_train]

print("Total images in versicolor directory:", len(ve
rsicolor_directory))
print(type(versicolor_directory))
# print(versicolor_directory, "\n")

print("Total images in versicolor train list:", versi
color_train_file_count)
print(type(versicolor_random_train))
# print(versicolor_random_train, "\n")

print("Total images in versicolor test list:", len(ve
rsicolor_random_test))
print(type(versicolor_random_test))
# print(versicolor_random_test)
```

```
Total images in versicolor directory: 50
<class 'list'>
Total images in versicolor train list: 3
5
<class 'list'>
Total images in versicolor test list: 15
<class 'list'>
```

And virginica.

In [14]:

```
# virginica
# from all images list, remove the training list, resu
lting in testing list
virginica_random_test = [x for x in virginica_directo
ry if x not in virginica_random_train]

print("Total images in virginica directory:", len(vir
ginica_directory))
print(type(virginica_directory))
# print(virginica_directory, "\n")

print("Total images in virginica train list:", virgin
ica_train_file_count)
print(type(virginica_random_train))
# print(virginica_random_train, "\n")

print("Total images in virginica test list:", len(vir
ginica_random_test))
print(type(virginica_random_test))
# print(virginica_random_test)
```

```
Total images in virginica directory: 50
<class 'list'>
Total images in virginica train list: 35
<class 'list'>
Total images in virginica test list: 15
<class 'list'>
```

All lists containing the names of the training and testing files to be copied are now defined.

Copy randomized image files to new folder locations

The next step is to copy the training images into their new folder locations.

In [15]:

```
# training
# setosa
files = setosa_random_train
moveFrom = orig_setosa_folder_path
moveTo = new_train_setosa_folder_path
```

```
for f in files:
    shutil.copy(moveFrom + f, moveTo)
#     print(f, "copied to new directory") # for testin
g


# versicolor
files = versicolor_random_train
moveFrom = orig_versicolor_folder_path
moveTo = new_train_versicolor_folder_path

for f in files:
    shutil.copy(moveFrom + f, moveTo)


# virginica
files = virginica_random_train
moveFrom = orig_virginica_folder_path
moveTo = new_train_virginica_folder_path

for f in files:
    shutil.copy(moveFrom + f, moveTo)
```

Now copy the testing files.

In [16]:

```
# testing

# setosa
files = setosa_random_test
moveFrom = orig_setosa_folder_path
moveTo = new_test_setosa_folder_path

for f in files:
    shutil.copy(moveFrom + f, moveTo)
#     print(f, "copied to new directory") # for testin
g


# versicolor
files = versicolor_random_test
moveFrom = orig_versicolor_folder_path
moveTo = new_test_versicolor_folder_path

for f in files:
    shutil.copy(moveFrom + f, moveTo)
```

```
# virginica
files = virginica_random_test
moveFrom = orig_virginica_folder_path
moveTo = new_test_virginica_folder_path

for f in files:
    shutil.copy(moveFrom + f, moveTo)
```

Check that everything worked correctly by counting the files in each folder. Life without a proper IDE is hard-knock.

In [17]:

```
# Check: count total images in folder

# setosa
# train
path, dirs, files = next(os.walk(new_train_setosa_fol
der_path))
file_count = len(files)
# float for math
file_count = int(file_count) # convert for later use
print("new setosa train file_count:", file_count)

# test
path, dirs, files = next(os.walk(new_test_setosa_fold
er_path))
file_count = len(files)
# float for math
file_count = int(file_count) # convert for later use
print("new setosa test file_count:", file_count, "\n"
)


# versicolor
# train
path, dirs, files = next(os.walk(new_train_versicolor
_folder_path))
file_count = len(files)
# float for math
file_count = int(file_count) # convert for later use
print("new versicolor train file_count:", file_count)

# test
path, dirs, files = next(os.walk(new_test_versicolor_
folder_path))
```

```
file_count = len(files)
# float for math
file_count = int(file_count) # convert for later use
print("new versicolor test file_count:", file_count,
"\n")


# virginica
# train
path, dirs, files = next(os.walk(new_train_virginica_
folder_path))
file_count = len(files)
# float for math
file_count = int(file_count) # convert for later use
print("new virginica train file_count:", file_count)

# test
path, dirs, files = next(os.walk(new_test_virginica_f
older_path))
file_count = len(files)
# float for math
file_count = int(file_count) # convert for later use
print("new virginica test file_count:", file_count)
```

```
new setosa train file_count: 35
new setosa test file_count: 15

new versicolor train file_count: 35
new versicolor test file_count: 15

new virginica train file_count: 35
new virginica test file_count: 15
```

Our folders are now full of content, and in the correct structure for processing.

Preview jpg images

Let's peak at our images to see what we're working with. Using Python Image Library (PIL), we can open a randomized (https://stackoverflow.com/questions/701402/best-way-to-choose-a-random-file-from-a-directory) image of each species class.

In [18]:

```
# load random image
# Load the images
# Image.open is from Import
```

```
from PIL import Image

print("random images selected:")

# setosa
setosa_image = random.choice(os.listdir(new_train_set
osa_folder_path)) #change dir name to whatever
setosa_image_path = orig_setosa_folder_path + setosa_
image
print(setosa_image_path)
setosa_load = Image.open(setosa_image_path)

# versicolor
versicolor_image = random.choice(os.listdir(new_train
_versicolor_folder_path)) #change dir name to whatever
versicolor_image_path = orig_versicolor_folder_path +
 versicolor_image
print(versicolor_image_path)
versicolor_load = Image.open(versicolor_image_path)

# virginica
virginica_image = random.choice(os.listdir(new_train_
virginica_folder_path)) #change dir name to whatever
virginica_image_path = orig_virginica_folder_path + v
irginica_image
print(virginica_image_path)
virginica_load = Image.open(virginica_image_path)
```

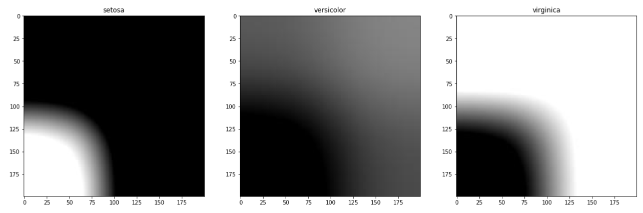
```
random images selected:
../input/iris-as-images/iris_imgs/Iris_I
mgs/setosa/Img_41.jpg
../input/iris-as-images/iris_imgs/Iris_I
mgs/versicolor/Img_92.jpg
../input/iris-as-images/iris_imgs/Iris_I
mgs/virginica/Img_103.jpg
```

In [19]:

```
import matplotlib.pyplot as plt

# plot images
f = plt.figure(figsize = (20, 10))

# add_subplot(nrows, ncols, index, **kwargs)
a1 = f.add_subplot(1, 3, 1) # (rows x columns x posit
ion)
img_plot = plt.imshow(setosa_load)
a1.set_title('setosa')
```

```
a2 = f.add_subplot(1, 3, 2)
img_plot = plt.imshow(versicolor_load)
a2.set_title('versicolor')

a3 = f.add_subplot(1, 3, 3)
img_plot = plt.imshow(virginica_load)
a3.set_title('virginica')

print("Note: Sometimes an image won't load. Seems err
adic.")
```

```
Note: Sometimes an image won't load. See
ms erradic.
```



These were csv files converted into images.

Load Data

Load Data Generator

In [20]:

```
from tensorflow.python.keras.preprocessing.image impo
rt ImageDataGenerator
from tensorflow.python.keras.applications.resnet50 im
port preprocess_input # required for resnet50

batch_size = 5   # impacts processing time and accurac
y. This image set is small, so it's not a memory issu
e.
image_size = 200 # image size is 200x200

# Don't horizontal flip this dataset, as it would chan
ge the dataset's meaning (i.e. don't flip letters eith
er!)
data_generator = ImageDataGenerator(preprocessing_fun
ction = preprocess_input)
```

Train Test Split

Train Test Split

Execute the split and label the datasets.

In [21]:

```
print(new_train_path)
training_generator    = data_generator.flow_from_direc
tory(
        new_train_path,
        target_size  = (image_size, image_size),
        batch_size   = batch_size,
        class_mode   = "categorical")
# shuffle      = False) # shuffles data at generation.
 Setting to False messes everything up.

print(new_test_path)
validation_generator = data_generator.flow_from_direc
tory(
        new_test_path,
        target_size  = (image_size, image_size),
        batch_size   = batch_size,
        class_mode   = "categorical")
# shuffle      = False) # shuffles data at generation.
 Setting to False messes everything up.
```

```
Iris_Imgs/train/
Found 105 images belonging to 3 classes.
Iris_Imgs/test/
Found 45 images belonging to 3 classes.
```

Capture and Set Hyperparameters

In [22]:

```
from keras.utils import to_categorical

# https://stackoverflow.com/questions/50186035/error-m
ulti-classification-neural-network-using-keras
# https://pastebin.com/V1YwJW3X

# train_labels = training_generator.classes # UNUSED
# test_labels = validation_generator.classes # UNUSED

num_classes = len(training_generator.class_indices) #
 setosa, versicolor, virginica
print("num_classes:", num_classes)

# Save the dictionary of labels to be used in train
# np.save('class_indicies.npy', training_generator.cla
ss_indices)

# Convert labels to categorical vectors - [1,0,0], [0,
1,0], [0,0,1]
# train_labels = to_categorical(train_labels, num_clas
ses = num_classes) # UNUSED
# test_labels = to_categorical(test_labels, num_classe
s = num_classes) # UNUSED

train_steps = training_generator.samples // batch_siz
e # / for floating point, // for int (python 3)
print("train_steps:", train_steps)
val_steps = validation_generator.samples // batch_siz
e
print("val_steps:", val_steps)
```

```
Using TensorFlow backend.


num_classes: 3
train_steps: 21
val_steps: 9
```

Specify Model

This segment uses as guidence Dan Becker's transfer learning tutorial (https://www.kaggle.com/dansbecker/transfer-learning/notebook). If you look him up, he has other extremely clear and helpful tutorials as well.

In [23]:

```
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense
from tensorflow.python.keras.layers import Flatten
from tensorflow.python.keras.layers import BatchNorma
lization
from tensorflow.python.keras.layers import GlobalAver
agePooling2D
from tensorflow.python.keras.applications import ResN
et50

resnet_weights_path = "../input/resnet50/resnet50_wei
ghts_tf_dim_ordering_tf_kernels_notop.h5"
```

Define the neural net's layers.

It's an ongoing debate (https://github.com/keras-team/keras/issues/1802#issuecomment-187966878) whether batch normalization should be before or after activation. Based on this test (https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md#batch-normalization), I would place batch normalization after. However, in this architecture, I only add a batch normalization after ResNet, as placing one before is nonsensical, as would be placing it after the final dense layer.

In [24]:

```
model = Sequential()
model.add(ResNet50(include_top = False, pooling = "av
g", weights = resnet_weights_path)) # include top mea
ns no need to pop?
model.add(BatchNormalization())
model.add(Dense(num_classes, activation = "softmax"))

# The first ResNet layer is already trained, so we fre
eze it. The weights are already good.
model.layers[0].trainable = False

# review the configurations of all the layers in this
 CNN.
model.summary()
```

```
________________________________________
_________________________
Layer (type)                 Output Shap
e              Param #
========================================
=========================
resnet50 (Model)             (None, 204
8)              23587712
________________________________________
_________________________
batch_normalization (BatchNo (None, 204
8)              8192
________________________________________
_________________________
dense (Dense)                (None, 3)
               6147
========================================
=========================
Total params: 23,602,051
Trainable params: 10,243
Non-trainable params: 23,591,808
________________________________________
_________________________
```

Model Summary:

In → ResNet50 → Batch Normalization → Dense → Out

Above we see the value of an imported neural network. ResNet50 comes out of the blackbox with with 2,358,7712 pre-trained weights (non-trainable parameters). Our batch normalization plus dense parameters = 10,243 / 2,358,7712 = 0.04% of the total. It will be interesting to see what this extremely small training can result in.

Compile Model

In [25]:

```
model.compile(optimizer = "sgd", loss = "categorical_
crossentropy", metrics = ["accuracy"])
print("Loss function: " + model.loss)
```

```
Loss function: categorical_crossentropy
```

Fit Model

Instantiate a datagenerator. It has parameters you can input, but this is a plain-vanilla approach. We will also time (https://stackoverflow.com/questions/7370801/measure-time-elapsed-in-python) how long it takes to fit the model.

- steps_per_epoch the number of batch iterations before a training epoch is considered finished. If you have a training set of fixed size you can ignore it may be useful if you have a huge data set or if you are generating random data augmentations on the fly.
- validation_steps similar to steps_per_epoch but on the validation data set instead on the training data. If you have the time to go through your whole validation data set I recommend to skip this parameter.

In [26]:

```
import time

# clock start
start = time.time()
print("timer started", "\n")

epochsCount  = 30
```

```
# stepsPerEpoch = 1 # use this for processing shortcut

# I lost many debugging hours because I forgot to assi
gn mode.fit_generator to a variable!!!
resNetModel = model.fit_generator(
        training_generator,
        steps_per_epoch  = train_steps,
        epochs           = epochsCount, # goes throug
h each raw images n = epochs_count
        validation_data  = validation_generator)

# clock end
print("\ntimer stopped")
end = time.time()
```

```
timer started

Epoch 1/30
21/21 [==============================] -
7s 344ms/step - loss: 0.6844 - acc: 0.75
24 - val_loss: 0.8749 - val_acc: 0.5556
Epoch 2/30
21/21 [==============================] -
2s 79ms/step - loss: 0.2999 - acc: 0.895
2 - val_loss: 0.8632 - val_acc: 0.7111
Epoch 3/30
21/21 [==============================] -
2s 80ms/step - loss: 0.3927 - acc: 0.876
2 - val_loss: 0.9769 - val_acc: 0.6000
Epoch 4/30
21/21 [==============================] -
2s 80ms/step - loss: 0.2540 - acc: 0.895
2 - val_loss: 1.1314 - val_acc: 0.5556
Epoch 5/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1517 - acc: 0.952
4 - val_loss: 1.2830 - val_acc: 0.5556
Epoch 6/30
21/21 [==============================] -
2s 81ms/step - loss: 0.1618 - acc: 0.952
4 - val_loss: 1.4414 - val_acc: 0.5111
Epoch 7/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1663 - acc: 0.933
3 - val_loss: 1.3309 - val_acc: 0.4889
Epoch 8/30
21/21 [==============================] -
2s 79ms/step - loss: 0.1889 - acc: 0.942
9 - val_loss: 1.5789 - val_acc: 0.4667
```

```
Epoch 9/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1761 - acc: 0.942
9 - val_loss: 1.5676 - val_acc: 0.4000
Epoch 10/30
21/21 [==============================] -
2s 79ms/step - loss: 0.1233 - acc: 0.952
4 - val_loss: 1.4843 - val_acc: 0.4444
Epoch 11/30
21/21 [==============================] -
2s 80ms/step - loss: 0.2269 - acc: 0.923
8 - val_loss: 2.0273 - val_acc: 0.3111
Epoch 12/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1267 - acc: 0.971
4 - val_loss: 1.8037 - val_acc: 0.3778
Epoch 13/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1194 - acc: 0.952
4 - val_loss: 1.7821 - val_acc: 0.3778
Epoch 14/30
21/21 [==============================] -
2s 79ms/step - loss: 0.2158 - acc: 0.942
9 - val_loss: 1.5674 - val_acc: 0.4889
Epoch 15/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1086 - acc: 0.971
4 - val_loss: 1.4788 - val_acc: 0.4667
Epoch 16/30
21/21 [==============================] -
2s 79ms/step - loss: 0.0925 - acc: 0.961
9 - val_loss: 1.4203 - val_acc: 0.5111
Epoch 17/30
21/21 [==============================] -
2s 79ms/step - loss: 0.1102 - acc: 0.971
4 - val_loss: 1.3026 - val_acc: 0.4889
Epoch 18/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1507 - acc: 0.952
4 - val_loss: 1.4192 - val_acc: 0.4889
Epoch 19/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1249 - acc: 0.971
4 - val_loss: 1.1965 - val_acc: 0.6000
Epoch 20/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1233 - acc: 0.952
4 - val_loss: 1.0285 - val_acc: 0.6222
Epoch 21/30
21/21 [==============================] -
2s 79ms/step - loss: 0.1827 - acc: 0.942
```

```
2s 79ms/step - loss: 0.1827 - acc: 0.942
9 - val_loss: 1.1160 - val_acc: 0.6444
Epoch 22/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1967 - acc: 0.971
4 - val_loss: 1.0533 - val_acc: 0.7111
Epoch 23/30
21/21 [==============================] -
2s 80ms/step - loss: 0.1601 - acc: 0.942
9 - val_loss: 0.6107 - val_acc: 0.8000
Epoch 24/30
21/21 [==============================] -
2s 79ms/step - loss: 0.2141 - acc: 0.952
4 - val_loss: 0.4873 - val_acc: 0.8222
Epoch 25/30
21/21 [==============================] -
2s 79ms/step - loss: 0.1636 - acc: 0.961
9 - val_loss: 0.1671 - val_acc: 0.9333
Epoch 26/30
21/21 [==============================] -
2s 81ms/step - loss: 0.1978 - acc: 0.914
3 - val_loss: 0.3226 - val_acc: 0.9111
Epoch 27/30
21/21 [==============================] -
2s 79ms/step - loss: 0.0372 - acc: 0.990
5 - val_loss: 0.1531 - val_acc: 0.9333
Epoch 28/30
21/21 [==============================] -
2s 81ms/step - loss: 0.1581 - acc: 0.933
3 - val_loss: 0.2620 - val_acc: 0.9333
Epoch 29/30
21/21 [==============================] -
2s 79ms/step - loss: 0.2385 - acc: 0.914
3 - val_loss: 0.1154 - val_acc: 0.9333
Epoch 30/30
21/21 [==============================] -
2s 79ms/step - loss: 0.0252 - acc: 1.000
0 - val_loss: 0.1211 - val_acc: 0.9778

timer stopped
```

Calculate time difference. Time per epoch varies with layer configuration, batch size, image size, steps per epoch and other factors. Relevant hyperparameter info can be found here (https://datascience.stackexchange.com/questions/29719/how-to-set-batch-size-steps-per-epoch-and-validation-steps).

In [27]:

```
time_to_fit = end - start
time_to_fit = round(time_to_fit, 5)
print("time (seconds) to fit:", time_to_fit)

time_per_epoch = time_to_fit / epochsCount
time_per_epoch = round(time_per_epoch, 5)
print("Average time (seconds) per epoch:", time_per_e
poch)
```

```
time (seconds) to fit: 55.82748
Average time (seconds) per epoch: 1.8609
2
```

Save the model for future use.

In [28]:

```
model.save("resNetModelSave")
```

Evaluate Model

Accuracy and Loss Chart

Evaluate the fit's performance. An example of plotting can be found here (https://www.kaggle.com/ottpeterr/convolutional-digit-classification-99-acc) and, for superior work, here (https://www.kaggle.com/dansbecker/transfer-learning).

In [29]:

```
acc      = resNetModel.history['acc']
val_acc  = resNetModel.history['val_acc']
loss     = resNetModel.history['loss']
val_loss = resNetModel.history['val_loss']
```

Use these values to create accuracy and loss plots.

In [30]:

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

plt.figure(figsize = (20, 10));
plt.subplot(2,1,1)
plt.plot(epochs, acc, color = 'b', linestyle = '-', l
abel = 'Training Accuracy')
plt.plot(epochs, val_acc, color = 'r', label = 'Valid
ation Accuracy')
```

Did you find this Kernel useful?
Show your appreciation with an upvote

0

Comments (0) Sort by Select...

Click here to enter a comment...

Similar Kernels

Flower Classification Model [Tensorflow]

Single_Language_Pro...

Convolutional Neural Network: Classify Iris

Facial Expression Recognition In Tensorflow

Ideas For Image Features And Image Quality