Human Protein Atlas Image Classification

Classify subcellular protein patterns in human cells - a multi-label, multi-class classification problem

James Golding

The Human Protein Atlas is a Sweden-based initiative aimed at mapping all human proteins in cells, tissues and organs.

The goal is to develop a model capable of classifying mixed patterns of proteins in confocal microscopy images. The Human Protein Atlas will use these models to build a tool integrated with their smart-microscopy system to identify a protein's location(s) from a high-throughput image.

The objective is to predict protein organelle localization labels for each sample. There are in total 28 different labels present in the dataset. The dataset itself is comprised of 27 different cell types of highly different morphology, which affect the protein patterns of the different organelles. All image samples are represented by four filters (stored as individual files), the protein of interest (green) plus three cellular landmarks: nucleus (blue), microtubules (red), endoplasmic reticulum (yellow). The green filter should hence be used to predict the label, and the other filters are used as references.

000a6c98-bb9b-11e8-b2b9-ac1f6b6435d0_blue (https://drive.google.com/file/d/1Vzm-vsLK3St8FaF0zoFLzvLxSMkyiBls/view?usp=sharing)

000a6c98-bb9b-11e8-b2b9-ac1f6b6435d0_red (https://drive.google.com/file/d/1W3nMUsiGSmRtTn89hdPBZ8MWGncAL1DZ/view?usp=sharing)

00a6c98-bb9b-11e8-b2b9-ac1f6b6435d0_yellow (https://drive.google.com/file/d/1W5tz1kvPnkkKiuUT5gWuAwl1ZXfQbAjP/view?usp=sharing)

000a6c98-bb9b-11e8-b2b9-ac1f6b6435d0_green (https://drive.google.com/file/d/1W04Vw3dVFO7vYsCV8TJUJXJNYKl0uT4u/view?usp=sharing)

000a6c98-bb9b-11e8-b2b9-ac1f6b6435d0_rgb (https://drive.google.com/file/d/1-1MLt69TtXbUtRw48EgV99LPnf5V-R_C/view?usp=sharing)

Provided for this problem are two versions of the same images, a scaled set of 512x512 PNG files or alternatively, full size original images (a mix of 2048x2048 and 3072x3072 TIFF files).

The labels are provided for each sample in a .csv file. Each image ID can have one or many labels associated with it. A sample is shown below.

```
Id                                      Target
00070df0-bbc3-11e8-b2bc-ac1f6b6435d0  16 0
000a6c98-bb9b-11e8-b2b9-ac1f6b6435d0  7 1 2 0
000a9596-bbc4-11e8-b2bc-ac1f6b6435d0  5
000c99ba-bba4-11e8-b2b9-ac1f6b6435d0  1
001838f8-bbca-11e8-b2bc-ac1f6b6435d0  18
001bcdd2-bbb2-11e8-b2ba-ac1f6b6435d0  0
0020af02-bbba-11e8-b2ba-ac1f6b6435d0  25 2
002679c2-bbb6-11e8-b2ba-ac1f6b6435d0  0
```

In [0]:
```
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[0]: '2.2.4'

Mount Your Google Drive to this Colab VM

In [0]:
```
from google.colab import drive
drive.mount('/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client
_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&
redirect_uri=urn%3Aietf%3Awg%3Aoauth%3A2.0%3Aoob&scope=email%20https%3A%2F%2F
www.googleapis.com%2Fauth%2Fdocs.test%20https%3A%2F%2Fwww.googleapis.com%2Fau
th%2Fdrive%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive.photos.readonly%
20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fpeopleapi.readonly&response_type=
code

Enter your authorization code:
..........
Mounted at /gdrive
```

Class Distribution Analysis

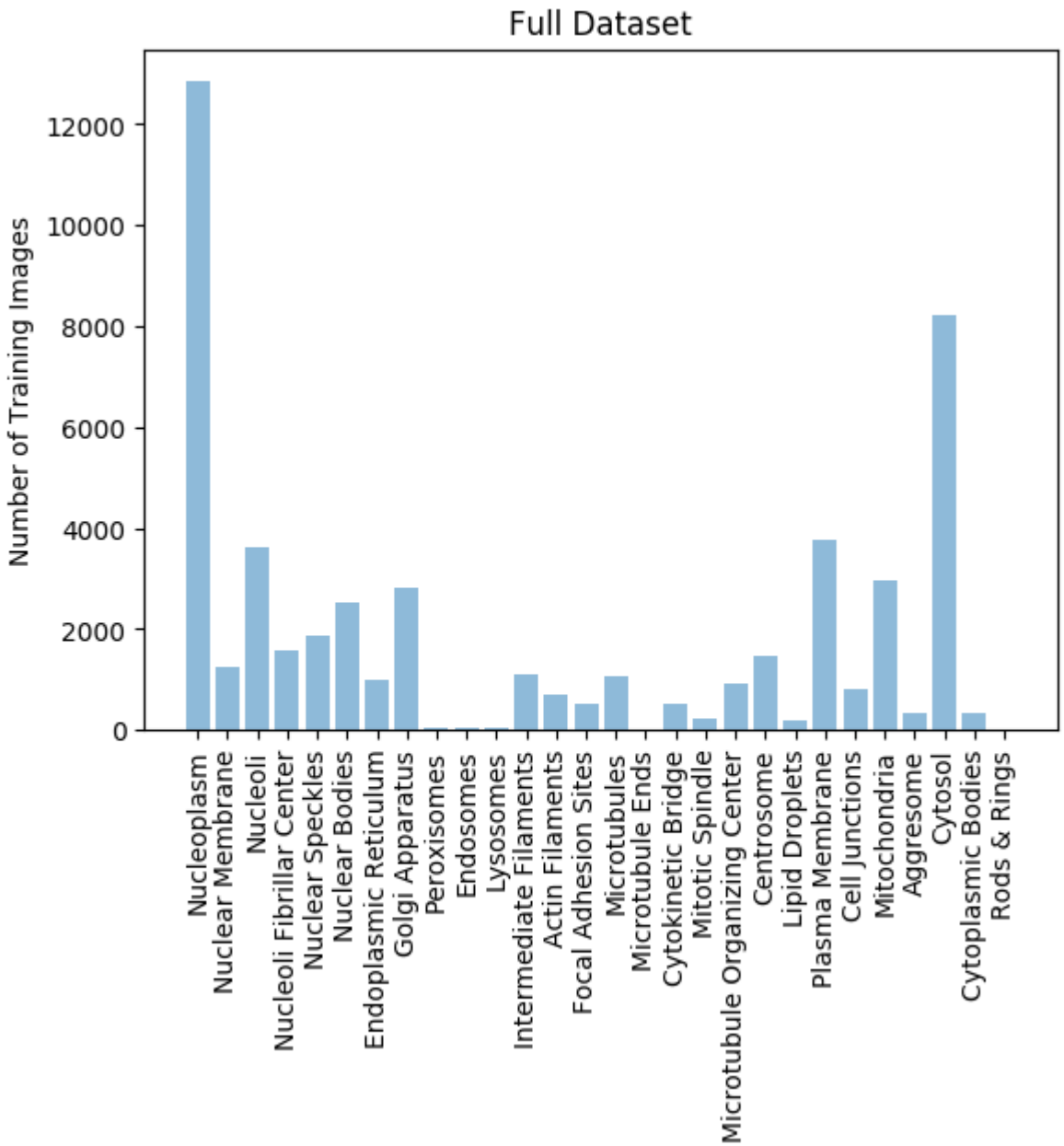
First look at the number of images available for training. The first graph below shows the number of training images for each class in the entire 31000 images provided. As can be seen, there are many classes which are quite sparse.

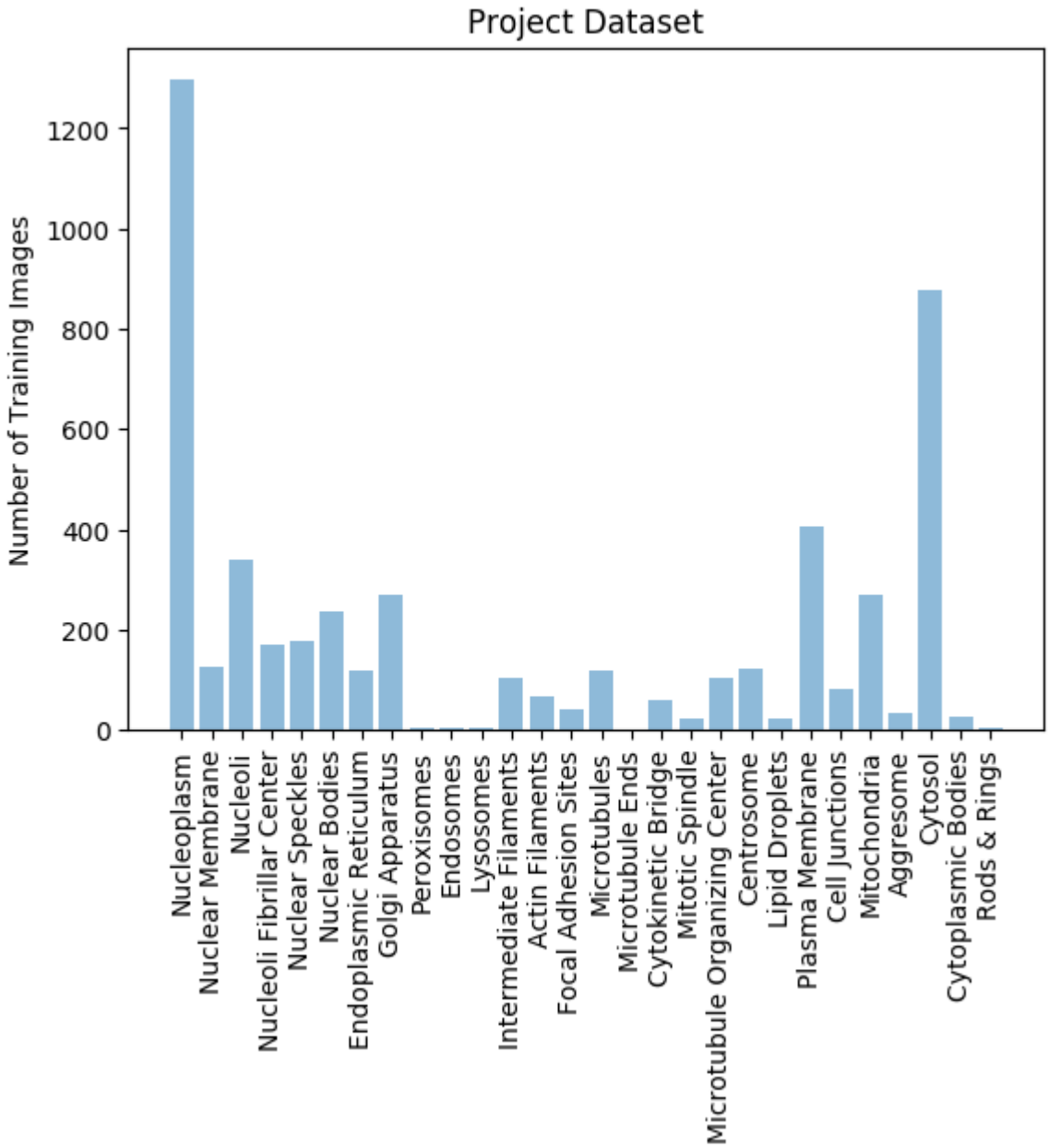
Due to RAM limitation on Colab, it was not possible to train on so many images! Colab limits to approximately 500 MB of RAM. As such, it was necessary to reduce the number of images used to ~4100, 3100 for training and 1000 for validation. To use this many, it was necessary to compress the images to further reduce file size which reduced resolution.

In the second graph, we see there are nearly 10 classes with fewer than 20 images available for training. Attempts could have been made to sort images and try to select more images for these classes or to try to eliminate them altogether, but time became a factor.

In [0]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt; plt.rcdefaults()
import matplotlib.pyplot as plt

num_images_full_dataset = 31000
num_images_term_project = 3100

# get the first n rows of labels. n is the number of images for training, vali
dation and testing
df = pd.read_csv('/gdrive/My Drive/DL_data/human_protein_atlas_original/train.
csv', nrows = num_images_full_dataset)

# Parse the training targets, then K-hot encode the 'targets' into 2D numpy ar
ray of floats
full_dataset_df = df[ :num_images_full_dataset]
label_list = full_dataset_df['Target'].str.split(" ", n = -1, expand = False).
as_matrix()
train_labels0 = np.zeros((28),dtype=int)
for i in range (0, num_images_full_dataset):
  for j in range (0,len(label_list[i])):
    label = int(label_list[i][j])
    train_labels0[label] += 1

project_df = df[ :num_images_term_project]
label_list = project_df['Target'].str.split(" ", n = -1, expand = False).as_ma
trix()
train_labels1 = np.zeros((28),dtype=int)
for i in range (0, num_images_term_project):
  for j in range (0,len(label_list[i])):
    label = int(label_list[i][j])
    train_labels1[label] += 1

objects = ('Nucleoplasm', 'Nuclear Membrane', 'Nucleoli', 'Nucleoli Fibrillar
 Center',
           'Nuclear Speckles', 'Nuclear Bodies', 'Endoplasmic Reticulum', 'Gol
gi Apparatus',
           'Peroxisomes', 'Endosomes', 'Lysosomes', 'Intermediate Filaments',
'Actin Filaments',
           'Focal Adhesion Sites', 'Microtubules', 'Microtubule Ends', 'Cytoki
netic Bridge',
           'Mitotic Spindle', 'Microtubule Organizing Center', 'Centrosome',
'Lipid Droplets',
           'Plasma Membrane', 'Cell Junctions', 'Mitochondria', 'Aggresome',
'Cytosol',
           'Cytoplasmic Bodies', 'Rods & Rings')
y_pos = np.arange(len(objects))
performance = train_labels0

plt.bar(y_pos, performance, align='center', alpha=0.5)
plt.xticks(y_pos, objects,rotation='vertical')
plt.ylabel('Number of Training Images')
plt.title('Full Dataset')

plt.show()
```

```
performance = train_labels1

plt.bar(y_pos, performance, align='center', alpha=0.5)
plt.xticks(y_pos, objects,rotation='vertical')
plt.ylabel('Number of Training Images')
plt.title('Project Dataset')

plt.show()
```


Full Dataset
Number of Training Images
0
2000
4000
6000
8000
10000
12000
Nucleoplasm
Nuclear Membrane
Nucleoli
Nucleoli Fibrillar Center
Nuclear Speckles
Nuclear Bodies
Endoplasmic Reticulum
Golgi Apparatus
Peroxisomes
Endosomes
Lysosomes
Intermediate Filaments
Actin Filaments
Focal Adhesion Sites
Microtubules
Microtubule Ends
Cytokinetic Bridge
Mitotic Spindle
Microtubule Organizing Center
Centrosome
Lipid Droplets
Plasma Membrane
Cell Junctions
Mitochondria
Aggresome
Cytosol
Cytoplasmic Bodies
Rods & Rings



In [0]:
```
!ls /gdrive/"My Drive"/DL_data/hpa_rgb_v1.0
#!ls /gdrive/"My Drive"/DL_data/human_protein_atlas_original
```

```
test  train  validation
```

Stack Images

Four gray scale images (labeled filename_yellow, filename_red, filename_blue, filename_green) for each label were provided for this competion. The competition rules state that predictions will be made on the 'green' images and that the 'red', 'blue' and 'yellow' images provide landmarks as to the location of the protein of interest.

First attempts were to train on the 'green' images as the contest suggests the others are only for reference. Initial results were not so good, but I was using 'accuracy' as a scoring metric when training on 'green' images. When using the accuracy metric, accuracy would go straight to 94% and increased very little.

Many Kagglers were discussing stacking the gray scale images into an RGB stack. Some other discussions considered an RGBA stack with yellow constituting the 'A'. As not much additional detail is obtained from the yellow image and due to the added complexity of feeding this stack to the CNN, it was decided to make an RGB stack of the images and use these for training and validation.

Further testing should be carried out to determine if training on just the 'green' images is a valid approach or if we see better results with the RGB stack.

The following code reads and merges the single channel gray scale images into a single RGB image, maintains the original file size and saves the newly created images into a new directory.

In [0]:
```
import pandas as pd
import os
import shutil
import cv2

# The path to the directory where the original dataset was uncompressed.
original_dataset_dir = '/gdrive/My Drive/DL_data/human_protein_atlas_original/
train'

# The directory where we will store the 'RGB' image stack
base_dir = '/gdrive/My Drive/DL_data/human_protein_atlas_rgb'
os.mkdir(base_dir)

# Read the .csv file containing the labels
labels = pd.read_csv('/gdrive/My Drive/DL_data/human_protein_atlas_original/tr
ain.csv')
num_images = len(labels)

# Merge the red, green and blue labeled images into a single RGB image and sav
e to the base_dir
for i in range(num_images):
  fn = labels['Id'][i]
  green = fn + '_green.png'
  g_image = cv2.imread(os.path.join(original_dataset_dir, green),0)
  blue = fn + '_blue.png'
  b_image = cv2.imread(os.path.join(original_dataset_dir, blue),0)
  red = fn + '_red.png'
  r_image = cv2.imread(os.path.join(original_dataset_dir, red),0)
  img = cv2.merge((b_image, g_image, r_image))
  cv2.imwrite(os.path.join(base_dir,fn + '_rgb.png'),img)
```

In [0]:
```
print('total number of images merged and transferred:', len(os.listdir(base_dir)))
```

Compress then copy RGB images into working directories

Copy stacked image files into a working directory consisting of training, validation and test folders. As this is a multi-label, multi-classification problem, images were placed directly in their respective folders and no subfolders were necessary for each class.

Colab allows approximately 500 MB of data to be uploaded and processed in RAM. First attempts at training used the full-sized 512 x 512 images. This however severely limited the number of images which could be uploaded to Colab to approximately 1400 of the 31000 images provided in the dataset. With this approach I reached a validation loss of about 0.15.

Compressing the images to 299 x 299, allowed for a little over 4100 images to be loaded into Colab's RAM. This approach gave better results during training with a best validation loss of 0.1242.

This size was chosen so that I could also try using a pretrained network, "InceptionResNetV2' and providing my own classifier on top. This attempt proved futile, with a best validation loss of only 0.2145. With these results, a train from scratch approach was carried out.

Tests with further data compression were not attempted.

In [0]:
```
import os
import pandas as pd
import shutil
import cv2

width = 299
height = 299

# The directory where the 'RGB' images are stored
rgb_image_dir = '/gdrive/My Drive/DL_data/human_protein_atlas_rgb'

# Create a new directory to store the compressed RGB images
base_dir = '/gdrive/My Drive/DL_data/hpa_sm_comp_rgb_v1.0'
os.mkdir(base_dir)

# Directories for our training, validation and test splits
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)

num_train_images = 3100
num_val_images = 1000
num_test_images = 50

# Get labels
labels = pd.read_csv('/gdrive/My Drive/DL_data/human_protein_atlas_original/tr
ain.csv')

# Now copy files from the 'RGB' image file directory into the respective train
ing, validation
# and testing directories. Note this is a small subset of the total images ava
ilable.
for i in range(0,num_train_images):
  fn = labels['Id'][i]
  image_name = fn + '_rgb.png'
  img = cv2.imread(os.path.join(rgb_image_dir, image_name),1)
  img = cv2.resize(img,(width,height))
  cv2.imwrite(os.path.join(train_dir,fn + '_rgb.png'),img)

for i in range(num_train_images,num_train_images+num_val_images):
  fn = labels['Id'][i]
  image_name = fn + '_rgb.png'
  img = cv2.imread(os.path.join(rgb_image_dir, image_name),1)
  img = cv2.resize(img,(width,height))
  cv2.imwrite(os.path.join(validation_dir,fn + '_rgb.png'),img)

for i in range(num_train_images+num_val_images,num_train_images+num_val_images
+num_test_images):
  fn = labels['Id'][i]
  image_name = fn + '_rgb.png'
  img = cv2.imread(os.path.join(rgb_image_dir, image_name),1)
  img = cv2.resize(img,(width,height))
  cv2.imwrite(os.path.join(test_dir,fn + '_rgb.png'),img)
```

Create the labels and vectorize the images

The labels were provided via a .csv file. The file was formatted such that the first column contained the image name and the second column indicated to which class or classes the image belonged.

```
Id                                      Target
00070df0-bbc3-11e8-b2bc-ac1f6b6435d0  16 0
000a6c98-bb9b-11e8-b2b9-ac1f6b6435d0  7 1 2 0
000a9596-bbc4-11e8-b2bc-ac1f6b6435d0  5
000c99ba-bba4-11e8-b2b9-ac1f6b6435d0  1
001838f8-bbca-11e8-b2bc-ac1f6b6435d0  18
001bcdd2-bbb2-11e8-b2ba-ac1f6b6435d0  0
0020af02-bbba-11e8-b2ba-ac1f6b6435d0  25 2
002679c2-bbb6-11e8-b2ba-ac1f6b6435d0  0
```

Three separate data frames were created, one each for training, validation and testing.

It was necessary to parse the labels and k-hot encode the labels in order to feed them into the network.

```
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0.],
 [1. 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0.],
 ...
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0.]]
```

This code also reads the images and preprocesses them into numpy arrays using the OpenCV library

Run this code every time I connect to Colab

In [0]:
```
import os
import pandas as pd
import shutil
import numpy as np
import cv2

def load_data():
  # The directory where the subset of 'RGB' images are mounted
  sm_rgb_dir = '/gdrive/My Drive/DL_data/hpa_sm_comp_rgb_v1.0'

  # Directories for our training, validation and test splits
  train_dir = os.path.join(sm_rgb_dir, 'train')
  validation_dir = os.path.join(sm_rgb_dir, 'validation')
  test_dir = os.path.join(sm_rgb_dir, 'test')

  num_train_images = len([f for f in os.listdir(train_dir) if os.path.isfile(o
s.path.join(train_dir, f))])
  num_validation_images = len([f for f in os.listdir(validation_dir) if os.pat
h.isfile(os.path.join(validation_dir, f))])
  num_test_images = len([f for f in os.listdir(test_dir) if os.path.isfile(os.
path.join(test_dir, f))])
  num_images = num_train_images + num_validation_images + num_test_images

  # get the first n rows of labels. n is the number of images for training, va
lidation and testing
  df = pd.read_csv('/gdrive/My Drive/DL_data/human_protein_atlas_original/trai
n.csv', nrows = num_images)

  # Parse the training targets, then K-hot encode the 'targets' into 2D numpy
 array of floats
  train_df = df[ :num_train_images]
  label_list = train_df['Target'].str.split(" ", n = -1, expand = False).as_ma
trix()
  train_labels = np.zeros((len(label_list),28),dtype=int)
  for i in range (0, num_train_images):
    for j in range (0,len(label_list[i])):
      label = int(label_list[i][j])
      train_labels[i][label] = 1

  # Parse the validation targets, then K-hot encode the 'targets' into 2D nump
y array of floats
  validation_df = df[num_train_images:num_train_images + num_validation_images
]
  label_list = validation_df['Target'].str.split(" ", n = -1, expand = False).
as_matrix()
  validation_labels = np.zeros((num_validation_images,28),dtype=int)
  for i in range(0,num_validation_images):
    for j in range (0,len(label_list[i])):
      label = int(label_list[i][j])
      validation_labels[i][label] = 1

  # Parse the test targets, then K-hot encode the 'targets' into 2D numpy arra
y of floats
  test_df = df[num_train_images + num_validation_images:num_images]
  label_list = test_df['Target'].str.split(" ", n = -1, expand = False).as_mat
rix()
```

```
  test_labels = np.zeros((num_test_images,28),dtype=int)
  for i in range(0,num_test_images):
    for j in range (0,len(label_list[i])):
      label = int(label_list[i][j])
      test_labels[i][label] = 1

  #train_images = np.zeros((len(train_labels),512,512,3)).astype('float32')
  #validation_images = np.zeros((len(validation_labels),512,512,3)).astype('fl
oat32')
  #test_images = np.zeros((len(test_labels),512,512,3)).astype('float32'

  train_images = np.zeros((len(train_labels),299,299,3)).astype('float32')
  validation_images = np.zeros((len(validation_labels),299,299,3)).astype('flo
at32')
  test_images = np.zeros((len(test_labels),299,299,3)).astype('float32')

  # convert the images to numpy arrays of 3 dimensions
  for i in range(0,num_train_images):
    fn = train_df['Id'][i]+'_rgb.png'
    img = cv2.imread(os.path.join(train_dir, fn),1) # 1 for rgb images
    train_images[i] = img

  for i in range(0,num_validation_images):
    fn = validation_df['Id'][i + num_train_images]+'_rgb.png'
    img = cv2.imread(os.path.join(validation_dir, fn),1) # 1 for rgb images
    validation_images[i] = img

  for i in range(0,num_test_images):
    fn = test_df['Id'][i + num_train_images + num_validation_images]+'_rgb.pn
g'
    img = cv2.imread(os.path.join(test_dir, fn),1) # 1 for rgb images
    test_images[i] = img

  return (train_images, train_labels, validation_images, validation_labels,
          test_images, test_labels);
```

In [0]:
```
train_images,train_labels,validation_images,validation_labels,test_images,test
_labels = load_data()
```

Build the image generator

In [0]:
```
from keras.preprocessing.image import ImageDataGenerator
train_batch_size = 25
validation_batch_size = 25
test_batch_size = 25

train_datagen = ImageDataGenerator(
  rescale = 1./255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = True,
  fill_mode = 'nearest')

test_datagen = ImageDataGenerator(rescale = 1./255)

train_generator = train_datagen.flow(
  train_images,
  train_labels,
  batch_size = train_batch_size)

validation_generator = test_datagen.flow(
  validation_images,
  validation_labels,
  batch_size = validation_batch_size)

test_generator = test_datagen.flow(
  test_images,
  batch_size = test_batch_size)
```

In [0]:
```
for data_batch, labels_batch in train_generator:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break
```

```
data batch shape: (25, 299, 299, 3)
labels batch shape: (25, 28)
```

In [0]:
```
print (train_images.shape)
print(train_images.dtype)
```

```
(3100, 299, 299, 3)
float32
```

Define custom metric for measuring success

Accuracy as a metric was not appropriate for this type of problem. During training, accuracy would go straight to 94% accuracy after just 1 epoch and hardly improve. This was true for both training and validation samples. Additionally, the Kaggle competition stated that an F1-score would be used as a scoring metric. Utilizing the f1 function, a typical incremental improvement in training accuracy was observed.

Keras use to provide an F1 scoring metric in earlier releases, but this was discontinued in later releases. As such, it was necessary to provide a custom metric. The f1 function below was borrowed from a fellow Kaggler competing in the 'Human Protein Image Classification' problem.

https://www.kaggle.com/guglielmocamporese/macro-f1-score-keras (https://www.kaggle.com/guglielmocamporese/macro-f1-score-keras)

F1 score is a statistical measurement that considers both the precision p and the recall r of the test to compute the score: p is the number of correct positive results divided by the number of all positive results returned by the classifier, and r is the number of correct positive results divided by the number of all relevant samples (all samples that should have been identified as positive). The F1 score is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.

https://en.wikipedia.org/wiki/F1_score (https://en.wikipedia.org/wiki/F1_score)

In [0]:
```
import tensorflow as tf
import keras.backend as K

def f1(y_true, y_pred):
    y_pred = K.round(y_pred)
    tp = K.sum(K.cast(y_true*y_pred, 'float'), axis=0)
    # tn = K.sum(K.cast((1-y_true)*(1-y_pred), 'float'), axis=0)
    fp = K.sum(K.cast((1-y_true)*y_pred, 'float'), axis=0)
    fn = K.sum(K.cast(y_true*(1-y_pred), 'float'), axis=0)

    p = tp / (tp + fp + K.epsilon())
    r = tp / (tp + fn + K.epsilon())

    f1 = 2*p*r / (p+r+K.epsilon())
    f1 = tf.where(tf.is_nan(f1), tf.zeros_like(f1), f1)
    return K.mean(f1)
```

Use smoothing function for graphing results

In [0]:
```
import matplotlib.pyplot as plt

def smooth_curve(points, factor=0.8):
  smoothed_points = []
  for point in points:
    if smoothed_points:
      previous = smoothed_points[-1]
      smoothed_points.append(previous * factor + point * (1 - factor))
    else:
      smoothed_points.append(point)
  return smoothed_points
```

Upload training weights saved locally on this pc

In [0]:
```
from google.colab import files

uploaded = files.upload()

for fn in uploaded.keys():
  print('User uploaded file "{name}" with length {length} bytes'.format(
      name=fn, length=len(uploaded[fn])))
```

Build the model

Several different sized models were tested in an attempt to identify which gave the best performance.

Overfitting was clearly an issue from the beginning due to there being so few images for many of the classes. Different sized networks were tested, including fewer layers and smaller number of neurons in the last Conv2D layer and the first dense layer. The smaller network did not perform as well as the network shown below.

Note that as this is a multi-label, multi-classifcation problem, used were sigmoid activation, binary_crossentropy loss functions and Adam optimizer. Categorical_crossentropy and softmax were not applicable to this type of problem.

Note we use the f1 scoring metric.

In [0]:
```
from keras import layers
from keras import models
from keras import optimizers

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(299, 299, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(512, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(512, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(28, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.Adam(lr=1e-4),
              metrics=[f1])
```

In [0]:
```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_11 (Conv2D)           (None, 297, 297, 32)      896
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 148, 148, 32)      0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 146, 146, 64)      18496
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 73, 73, 64)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 71, 71, 128)       73856
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 35, 35, 128)       0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 33, 33, 256)       295168
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 14, 14, 512)       1180160
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 7, 7, 512)         0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 5, 5, 512)         2359808
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 2, 2, 512)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 2048)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 2048)              0
_________________________________________________________________
dense_5 (Dense)              (None, 512)               1049088
_________________________________________________________________
dense_6 (Dense)              (None, 28)                14364
=================================================================
Total params: 4,991,836
Trainable params: 4,991,836
Non-trainable params: 0
_________________________________________________________________
```

Train the network

In [0]:
```
num_train_images = 3100
num_val_images = 1000
num_test_images = 50

history = model.fit_generator(
      train_generator,
      steps_per_epoch = num_train_images/train_batch_size, # 124 = 3100 / 25
      epochs=125,
      validation_data=validation_generator,
      validation_steps=num_val_images/validation_batch_size) # 40 = 1000 / 25
model.save('hpa_v1.1.h3')
```

```
Epoch 1/125
124/124 [==============================] - 64s 518ms/step - loss: 0.2431 - f
1: 0.0134 - val_loss: 0.1789 - val_f1: 0.0000e+00
Epoch 2/125
124/124 [==============================] - 63s 506ms/step - loss: 0.1807 - f
1: 0.0093 - val_loss: 0.1763 - val_f1: 0.0000e+00
Epoch 3/125
124/124 [==============================] - 63s 511ms/step - loss: 0.1790 - f
1: 0.0073 - val_loss: 0.1760 - val_f1: 0.0000e+00
Epoch 4/125
124/124 [==============================] - 65s 525ms/step - loss: 0.1772 - f
1: 0.0063 - val_loss: 0.1806 - val_f1: 0.0000e+00
Epoch 5/125
124/124 [==============================] - 67s 544ms/step - loss: 0.1762 - f
1: 0.0101 - val_loss: 0.1681 - val_f1: 0.0160
Epoch 6/125
124/124 [==============================] - 67s 540ms/step - loss: 0.1707 - f
1: 0.0196 - val_loss: 0.1709 - val_f1: 0.0121
Epoch 7/125
124/124 [==============================] - 68s 547ms/step - loss: 0.1708 - f
1: 0.0188 - val_loss: 0.1667 - val_f1: 0.0253
Epoch 8/125
124/124 [==============================] - 65s 524ms/step - loss: 0.1684 - f
1: 0.0220 - val_loss: 0.1707 - val_f1: 0.0280
Epoch 9/125
124/124 [==============================] - 63s 507ms/step - loss: 0.1659 - f
1: 0.0292 - val_loss: 0.1618 - val_f1: 0.0414
Epoch 10/125
124/124 [==============================] - 65s 525ms/step - loss: 0.1621 - f
1: 0.0358 - val_loss: 0.1667 - val_f1: 0.0403
Epoch 11/125
124/124 [==============================] - 66s 530ms/step - loss: 0.1603 - f
1: 0.0390 - val_loss: 0.1606 - val_f1: 0.0478
Epoch 12/125
124/124 [==============================] - 65s 527ms/step - loss: 0.1596 - f
1: 0.0362 - val_loss: 0.1577 - val_f1: 0.0433
Epoch 13/125
124/124 [==============================] - 66s 529ms/step - loss: 0.1584 - f
1: 0.0375 - val_loss: 0.1600 - val_f1: 0.0342
Epoch 14/125
124/124 [==============================] - 66s 531ms/step - loss: 0.1570 - f
1: 0.0393 - val_loss: 0.1599 - val_f1: 0.0345
Epoch 15/125
124/124 [==============================] - 67s 538ms/step - loss: 0.1568 - f
1: 0.0407 - val_loss: 0.1590 - val_f1: 0.0440
Epoch 16/125
124/124 [==============================] - 66s 535ms/step - loss: 0.1561 - f
1: 0.0409 - val_loss: 0.1548 - val_f1: 0.0320
Epoch 17/125
124/124 [==============================] - 66s 536ms/step - loss: 0.1555 - f
1: 0.0400 - val_loss: 0.1613 - val_f1: 0.0416
Epoch 18/125
124/124 [==============================] - 66s 529ms/step - loss: 0.1547 - f
1: 0.0409 - val_loss: 0.1569 - val_f1: 0.0363
Epoch 19/125
124/124 [==============================] - 66s 534ms/step - loss: 0.1552 - f
1: 0.0407 - val_loss: 0.1534 - val_f1: 0.0431
```

```
Epoch 20/125
124/124 [==============================] - 66s 532ms/step - loss: 0.1545 - f
1: 0.0427 - val_loss: 0.1513 - val_f1: 0.0521
Epoch 21/125
124/124 [==============================] - 63s 507ms/step - loss: 0.1543 - f
1: 0.0429 - val_loss: 0.1544 - val_f1: 0.0449
Epoch 22/125
124/124 [==============================] - 64s 516ms/step - loss: 0.1524 - f
1: 0.0425 - val_loss: 0.1542 - val_f1: 0.0437
Epoch 23/125
124/124 [==============================] - 65s 528ms/step - loss: 0.1532 - f
1: 0.0422 - val_loss: 0.1548 - val_f1: 0.0472
Epoch 24/125
124/124 [==============================] - 66s 531ms/step - loss: 0.1512 - f
1: 0.0443 - val_loss: 0.1536 - val_f1: 0.0453
Epoch 25/125
124/124 [==============================] - 66s 536ms/step - loss: 0.1525 - f
1: 0.0458 - val_loss: 0.1571 - val_f1: 0.0449
Epoch 26/125
124/124 [==============================] - 66s 534ms/step - loss: 0.1512 - f
1: 0.0471 - val_loss: 0.1479 - val_f1: 0.0502
Epoch 27/125
124/124 [==============================] - 66s 531ms/step - loss: 0.1492 - f
1: 0.0510 - val_loss: 0.1480 - val_f1: 0.0576
Epoch 28/125
124/124 [==============================] - 66s 532ms/step - loss: 0.1492 - f
1: 0.0516 - val_loss: 0.1523 - val_f1: 0.0503
Epoch 29/125
124/124 [==============================] - 67s 538ms/step - loss: 0.1483 - f
1: 0.0531 - val_loss: 0.1514 - val_f1: 0.0744
Epoch 30/125
124/124 [==============================] - 66s 535ms/step - loss: 0.1486 - f
1: 0.0550 - val_loss: 0.1484 - val_f1: 0.0625
Epoch 31/125
124/124 [==============================] - 66s 528ms/step - loss: 0.1464 - f
1: 0.0585 - val_loss: 0.1438 - val_f1: 0.0635
Epoch 32/125
124/124 [==============================] - 65s 528ms/step - loss: 0.1465 - f
1: 0.0614 - val_loss: 0.1449 - val_f1: 0.0592
Epoch 33/125
124/124 [==============================] - 67s 542ms/step - loss: 0.1440 - f
1: 0.0605 - val_loss: 0.1510 - val_f1: 0.0728
Epoch 34/125
124/124 [==============================] - 70s 562ms/step - loss: 0.1463 - f
1: 0.0622 - val_loss: 0.1428 - val_f1: 0.0695
Epoch 35/125
124/124 [==============================] - 69s 559ms/step - loss: 0.1448 - f
1: 0.0659 - val_loss: 0.1453 - val_f1: 0.0761
Epoch 36/125
124/124 [==============================] - 69s 560ms/step - loss: 0.1441 - f
1: 0.0645 - val_loss: 0.1482 - val_f1: 0.0815
Epoch 37/125
124/124 [==============================] - 74s 595ms/step - loss: 0.1428 - f
1: 0.0698 - val_loss: 0.1463 - val_f1: 0.0708
Epoch 38/125
124/124 [==============================] - 75s 606ms/step - loss: 0.1437 - f
1: 0.0707 - val_loss: 0.1426 - val_f1: 0.0809
```

```
Epoch 39/125
124/124 [==============================] - 72s 584ms/step - loss: 0.1421 - f
1: 0.0720 - val_loss: 0.1436 - val_f1: 0.0706
Epoch 40/125
124/124 [==============================] - 69s 553ms/step - loss: 0.1411 - f
1: 0.0768 - val_loss: 0.1408 - val_f1: 0.0762
Epoch 41/125
124/124 [==============================] - 69s 555ms/step - loss: 0.1407 - f
1: 0.0784 - val_loss: 0.1493 - val_f1: 0.0819
Epoch 42/125
124/124 [==============================] - 69s 556ms/step - loss: 0.1420 - f
1: 0.0747 - val_loss: 0.1399 - val_f1: 0.0810
Epoch 43/125
124/124 [==============================] - 73s 588ms/step - loss: 0.1399 - f
1: 0.0805 - val_loss: 0.1454 - val_f1: 0.0885
Epoch 44/125
124/124 [==============================] - 74s 599ms/step - loss: 0.1394 - f
1: 0.0852 - val_loss: 0.1423 - val_f1: 0.0865
Epoch 45/125
124/124 [==============================] - 73s 592ms/step - loss: 0.1379 - f
1: 0.0824 - val_loss: 0.1445 - val_f1: 0.0815
Epoch 46/125
124/124 [==============================] - 70s 561ms/step - loss: 0.1373 - f
1: 0.0874 - val_loss: 0.1429 - val_f1: 0.0859
Epoch 47/125
124/124 [==============================] - 69s 560ms/step - loss: 0.1378 - f
1: 0.0876 - val_loss: 0.1478 - val_f1: 0.0714
Epoch 48/125
124/124 [==============================] - 69s 557ms/step - loss: 0.1364 - f
1: 0.0892 - val_loss: 0.1413 - val_f1: 0.0899
Epoch 49/125
124/124 [==============================] - 71s 572ms/step - loss: 0.1351 - f
1: 0.0930 - val_loss: 0.1394 - val_f1: 0.0840
Epoch 50/125
124/124 [==============================] - 75s 603ms/step - loss: 0.1355 - f
1: 0.0900 - val_loss: 0.1468 - val_f1: 0.0896
Epoch 51/125
124/124 [==============================] - 75s 602ms/step - loss: 0.1346 - f
1: 0.0922 - val_loss: 0.1351 - val_f1: 0.0888
Epoch 52/125
124/124 [==============================] - 70s 563ms/step - loss: 0.1350 - f
1: 0.0952 - val_loss: 0.1445 - val_f1: 0.1050
Epoch 53/125
124/124 [==============================] - 70s 562ms/step - loss: 0.1335 - f
1: 0.0983 - val_loss: 0.1355 - val_f1: 0.0964
Epoch 54/125
124/124 [==============================] - 69s 560ms/step - loss: 0.1328 - f
1: 0.0989 - val_loss: 0.1371 - val_f1: 0.1138
Epoch 55/125
124/124 [==============================] - 72s 577ms/step - loss: 0.1314 - f
1: 0.1033 - val_loss: 0.1348 - val_f1: 0.1055
Epoch 56/125
124/124 [==============================] - 76s 611ms/step - loss: 0.1322 - f
1: 0.1045 - val_loss: 0.1367 - val_f1: 0.1045
Epoch 57/125
124/124 [==============================] - 76s 609ms/step - loss: 0.1304 - f
1: 0.1029 - val_loss: 0.1426 - val_f1: 0.1014
```

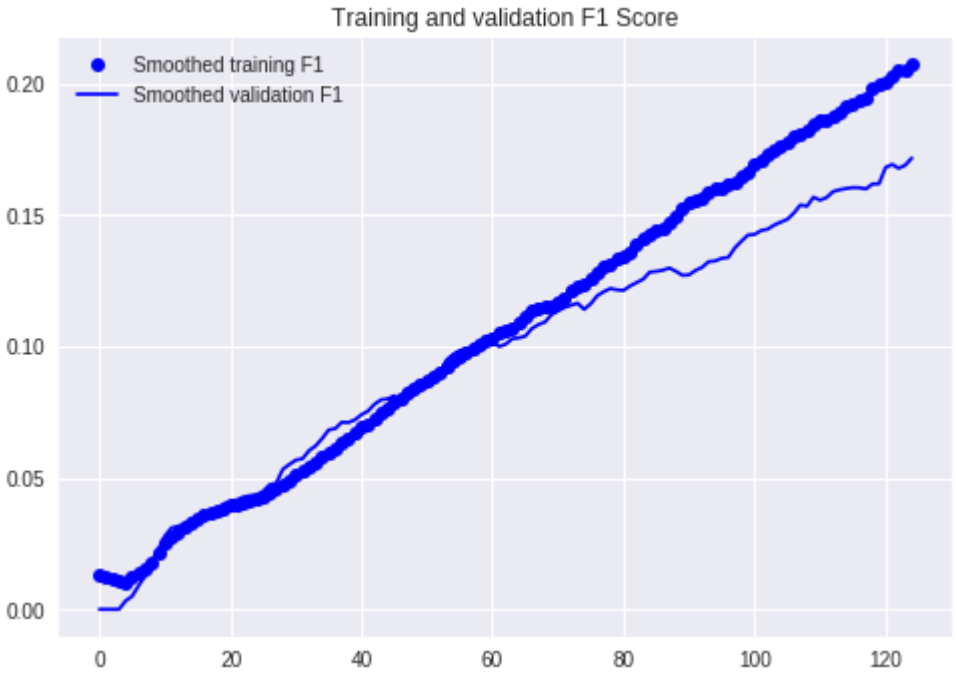
```
Epoch 58/125
124/124 [==============================] - 68s 547ms/step - loss: 0.1309 - f
1: 0.1040 - val_loss: 0.1372 - val_f1: 0.1006
Epoch 59/125
124/124 [==============================] - 68s 548ms/step - loss: 0.1303 - f
1: 0.1065 - val_loss: 0.1389 - val_f1: 0.0994
Epoch 60/125
124/124 [==============================] - 68s 545ms/step - loss: 0.1296 - f
1: 0.1083 - val_loss: 0.1356 - val_f1: 0.1136
Epoch 61/125
124/124 [==============================] - 65s 521ms/step - loss: 0.1301 - f
1: 0.1087 - val_loss: 0.1347 - val_f1: 0.1008
Epoch 62/125
124/124 [==============================] - 62s 501ms/step - loss: 0.1285 - f
1: 0.1116 - val_loss: 0.1400 - val_f1: 0.0924
Epoch 63/125
124/124 [==============================] - 61s 495ms/step - loss: 0.1287 - f
1: 0.1098 - val_loss: 0.1377 - val_f1: 0.1043
Epoch 64/125
124/124 [==============================] - 62s 502ms/step - loss: 0.1284 - f
1: 0.1117 - val_loss: 0.1368 - val_f1: 0.1113
Epoch 65/125
124/124 [==============================] - 62s 504ms/step - loss: 0.1261 - f
1: 0.1188 - val_loss: 0.1335 - val_f1: 0.1047
Epoch 66/125
124/124 [==============================] - 60s 486ms/step - loss: 0.1264 - f
1: 0.1212 - val_loss: 0.1381 - val_f1: 0.1066
Epoch 67/125
124/124 [==============================] - 61s 489ms/step - loss: 0.1262 - f
1: 0.1201 - val_loss: 0.1356 - val_f1: 0.1183
Epoch 68/125
124/124 [==============================] - 61s 490ms/step - loss: 0.1245 - f
1: 0.1206 - val_loss: 0.1362 - val_f1: 0.1146
Epoch 69/125
124/124 [==============================] - 61s 495ms/step - loss: 0.1258 - f
1: 0.1161 - val_loss: 0.1345 - val_f1: 0.1131
Epoch 70/125
124/124 [==============================] - 62s 498ms/step - loss: 0.1244 - f
1: 0.1148 - val_loss: 0.1357 - val_f1: 0.1236
Epoch 71/125
124/124 [==============================] - 60s 488ms/step - loss: 0.1241 - f
1: 0.1233 - val_loss: 0.1353 - val_f1: 0.1192
Epoch 72/125
124/124 [==============================] - 60s 486ms/step - loss: 0.1233 - f
1: 0.1242 - val_loss: 0.1307 - val_f1: 0.1201
Epoch 73/125
124/124 [==============================] - 61s 494ms/step - loss: 0.1209 - f
1: 0.1355 - val_loss: 0.1389 - val_f1: 0.1192
Epoch 74/125
124/124 [==============================] - 60s 488ms/step - loss: 0.1226 - f
1: 0.1298 - val_loss: 0.1332 - val_f1: 0.1196
Epoch 75/125
124/124 [==============================] - 62s 498ms/step - loss: 0.1207 - f
1: 0.1269 - val_loss: 0.1363 - val_f1: 0.1049
Epoch 76/125
124/124 [==============================] - 61s 493ms/step - loss: 0.1212 - f
1: 0.1355 - val_loss: 0.1373 - val_f1: 0.1249
```

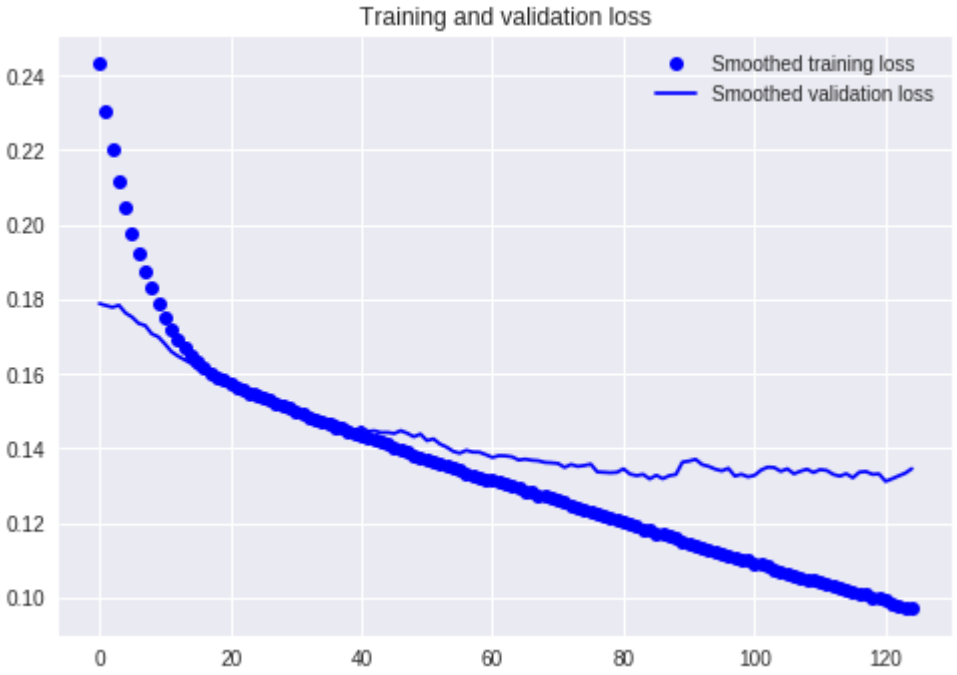
```
Epoch 77/125
124/124 [==============================] - 60s 483ms/step - loss: 0.1203 - f
1: 0.1355 - val_loss: 0.1254 - val_f1: 0.1315
Epoch 78/125
124/124 [==============================] - 59s 473ms/step - loss: 0.1195 - f
1: 0.1385 - val_loss: 0.1332 - val_f1: 0.1271
Epoch 79/125
124/124 [==============================] - 58s 471ms/step - loss: 0.1199 - f
1: 0.1338 - val_loss: 0.1330 - val_f1: 0.1269
Epoch 80/125
124/124 [==============================] - 62s 496ms/step - loss: 0.1183 - f
1: 0.1456 - val_loss: 0.1341 - val_f1: 0.1196
Epoch 81/125
124/124 [==============================] - 64s 519ms/step - loss: 0.1187 - f
1: 0.1374 - val_loss: 0.1383 - val_f1: 0.1205
Epoch 82/125
124/124 [==============================] - 64s 513ms/step - loss: 0.1175 - f
1: 0.1414 - val_loss: 0.1275 - val_f1: 0.1298
Epoch 83/125
124/124 [==============================] - 64s 519ms/step - loss: 0.1160 - f
1: 0.1489 - val_loss: 0.1311 - val_f1: 0.1295
Epoch 84/125
124/124 [==============================] - 65s 525ms/step - loss: 0.1155 - f
1: 0.1515 - val_loss: 0.1344 - val_f1: 0.1309
Epoch 85/125
124/124 [==============================] - 72s 581ms/step - loss: 0.1164 - f
1: 0.1501 - val_loss: 0.1270 - val_f1: 0.1391
Epoch 86/125
124/124 [==============================] - 71s 575ms/step - loss: 0.1149 - f
1: 0.1499 - val_loss: 0.1368 - val_f1: 0.1298
Epoch 87/125
124/124 [==============================] - 71s 569ms/step - loss: 0.1162 - f
1: 0.1487 - val_loss: 0.1282 - val_f1: 0.1308
Epoch 88/125
124/124 [==============================] - 66s 532ms/step - loss: 0.1141 - f
1: 0.1550 - val_loss: 0.1360 - val_f1: 0.1332
Epoch 89/125
124/124 [==============================] - 65s 526ms/step - loss: 0.1130 - f
1: 0.1595 - val_loss: 0.1342 - val_f1: 0.1238
Epoch 90/125
124/124 [==============================] - 63s 508ms/step - loss: 0.1119 - f
1: 0.1645 - val_loss: 0.1497 - val_f1: 0.1211
Epoch 91/125
124/124 [==============================] - 63s 510ms/step - loss: 0.1118 - f
1: 0.1629 - val_loss: 0.1377 - val_f1: 0.1279
Epoch 92/125
124/124 [==============================] - 68s 552ms/step - loss: 0.1118 - f
1: 0.1592 - val_loss: 0.1390 - val_f1: 0.1358
Epoch 93/125
124/124 [==============================] - 71s 576ms/step - loss: 0.1119 - f
1: 0.1607 - val_loss: 0.1300 - val_f1: 0.1349
Epoch 94/125
124/124 [==============================] - 72s 580ms/step - loss: 0.1106 - f
1: 0.1668 - val_loss: 0.1333 - val_f1: 0.1409
Epoch 95/125
124/124 [==============================] - 69s 558ms/step - loss: 0.1089 - f
1: 0.1657 - val_loss: 0.1315 - val_f1: 0.1338
```

```
Epoch 96/125
124/124 [==============================] - 65s 525ms/step - loss: 0.1116 - f
1: 0.1617 - val_loss: 0.1322 - val_f1: 0.1376
Epoch 97/125
124/124 [==============================] - 64s 512ms/step - loss: 0.1083 - f
1: 0.1688 - val_loss: 0.1375 - val_f1: 0.1352
Epoch 98/125
124/124 [==============================] - 66s 529ms/step - loss: 0.1095 - f
1: 0.1659 - val_loss: 0.1242 - val_f1: 0.1522
Epoch 99/125
124/124 [==============================] - 65s 520ms/step - loss: 0.1082 - f
1: 0.1726 - val_loss: 0.1351 - val_f1: 0.1503
Epoch 100/125
124/124 [==============================] - 72s 580ms/step - loss: 0.1079 - f
1: 0.1745 - val_loss: 0.1296 - val_f1: 0.1518
Epoch 101/125
124/124 [==============================] - 72s 579ms/step - loss: 0.1051 - f
1: 0.1803 - val_loss: 0.1345 - val_f1: 0.1437
Epoch 102/125
124/124 [==============================] - 72s 581ms/step - loss: 0.1085 - f
1: 0.1765 - val_loss: 0.1399 - val_f1: 0.1500
Epoch 103/125
124/124 [==============================] - 65s 521ms/step - loss: 0.1062 - f
1: 0.1811 - val_loss: 0.1381 - val_f1: 0.1468
Epoch 104/125
124/124 [==============================] - 60s 485ms/step - loss: 0.1048 - f
1: 0.1825 - val_loss: 0.1345 - val_f1: 0.1526
Epoch 105/125
124/124 [==============================] - 58s 465ms/step - loss: 0.1043 - f
1: 0.1815 - val_loss: 0.1298 - val_f1: 0.1519
Epoch 106/125
124/124 [==============================] - 61s 489ms/step - loss: 0.1043 - f
1: 0.1858 - val_loss: 0.1373 - val_f1: 0.1527
Epoch 107/125
124/124 [==============================] - 66s 532ms/step - loss: 0.1039 - f
1: 0.1883 - val_loss: 0.1279 - val_f1: 0.1603
Epoch 108/125
124/124 [==============================] - 64s 517ms/step - loss: 0.1043 - f
1: 0.1837 - val_loss: 0.1366 - val_f1: 0.1664
Epoch 109/125
124/124 [==============================] - 72s 578ms/step - loss: 0.1028 - f
1: 0.1878 - val_loss: 0.1367 - val_f1: 0.1515
Epoch 110/125
124/124 [==============================] - 68s 548ms/step - loss: 0.1028 - f
1: 0.1935 - val_loss: 0.1300 - val_f1: 0.1706
Epoch 111/125
124/124 [==============================] - 66s 534ms/step - loss: 0.1022 - f
1: 0.1924 - val_loss: 0.1375 - val_f1: 0.1511
Epoch 112/125
124/124 [==============================] - 66s 536ms/step - loss: 0.1022 - f
1: 0.1869 - val_loss: 0.1333 - val_f1: 0.1608
Epoch 113/125
124/124 [==============================] - 69s 558ms/step - loss: 0.1015 - f
1: 0.1953 - val_loss: 0.1288 - val_f1: 0.1682
Epoch 114/125
124/124 [==============================] - 72s 577ms/step - loss: 0.1005 - f
1: 0.1928 - val_loss: 0.1307 - val_f1: 0.1625
```

```
Epoch 115/125
124/124 [==============================] - 66s 530ms/step - loss: 0.1006 - f
1: 0.2005 - val_loss: 0.1359 - val_f1: 0.1621
Epoch 116/125
124/124 [==============================] - 67s 543ms/step - loss: 0.0993 - f
1: 0.1950 - val_loss: 0.1278 - val_f1: 0.1619
Epoch 117/125
124/124 [==============================] - 67s 539ms/step - loss: 0.0987 - f
1: 0.2028 - val_loss: 0.1397 - val_f1: 0.1602
Epoch 118/125
124/124 [==============================] - 75s 607ms/step - loss: 0.0999 - f
1: 0.1958 - val_loss: 0.1345 - val_f1: 0.1582
Epoch 119/125
124/124 [==============================] - 66s 533ms/step - loss: 0.0974 - f
1: 0.2144 - val_loss: 0.1300 - val_f1: 0.1695
Epoch 120/125
124/124 [==============================] - 68s 549ms/step - loss: 0.0981 - f
1: 0.2039 - val_loss: 0.1336 - val_f1: 0.1623
Epoch 121/125
124/124 [==============================] - 73s 586ms/step - loss: 0.0980 - f
1: 0.2045 - val_loss: 0.1232 - val_f1: 0.1930
Epoch 122/125
124/124 [==============================] - 66s 534ms/step - loss: 0.0947 - f
1: 0.2133 - val_loss: 0.1347 - val_f1: 0.1740
Epoch 123/125
124/124 [==============================] - 67s 539ms/step - loss: 0.0957 - f
1: 0.2121 - val_loss: 0.1358 - val_f1: 0.1620
Epoch 124/125
124/124 [==============================] - 67s 543ms/step - loss: 0.0958 - f
1: 0.2065 - val_loss: 0.1363 - val_f1: 0.1738
Epoch 125/125
124/124 [==============================] - 71s 574ms/step - loss: 0.0958 - f
1: 0.2152 - val_loss: 0.1394 - val_f1: 0.1825
```

In [0]:
```
acc = history.history['f1']
val_acc = history.history['val_f1']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs,
         smooth_curve(acc), 'bo', label='Smoothed training F1')
plt.plot(epochs,
         smooth_curve(val_acc), 'b', label='Smoothed validation F1')
plt.title('Training and validation F1 Score')
plt.legend()

plt.figure()

plt.plot(epochs,
         smooth_curve(loss), 'bo', label='Smoothed training loss')
plt.plot(epochs,
         smooth_curve(val_loss), 'b', label='Smoothed validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```


Training and validation F1 Score
Smoothed training F1
Smoothed validation F1
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
120


Training and validation loss
Smoothed training loss
Smoothed validation loss
0.24
0.22
0.20
0.18
0.16
0.14
0.12
0.10
0
20
40
60
80
100
120

Sanity Check

Check to make sure that the network is making multiple classifications when appropriate and an idea of the accuracy. In test cases of images the network has never seen before, I see in about 20-25% of the cases a perfect classification. The network classifies one class correctly in approximately 50% of the remaining test images, but mis-classifies a second or third class. For the other 25% of the images, the network mis-classifies completely.

In the entire 31000 image dataset, 3 of the 28 classes had 28 or fewer images. The small subset of images used to train the model had 8 classes with fewer than 24 images each. Attempts to filter the images and ensure that as many images as possible were available for training and validation was not made as time became a limiting factor. Doing so should have improved the scoring.

Below is shown the output of 50 images the network has never seen before and compares the ground truth vs predicted.

In [0]:
```
y= (model.predict(test_images).round())
for i in range(0,len(y)):
  print ('actual   ', test_labels[i])
  print ('predicted', y[i].astype(int))
  print()
```

```
actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
predicted [0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

actual    [1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
predicted [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
```

```
actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]

actual    [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

actual    [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
actual    [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

actual    [1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

actual    [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
predicted [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

actual    [1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

In [0]:
```
from google.colab import files

files.download('hpa_v1.1.h3')
!ls
```

```
hpa_rgb_comp_v1.0.h1  hpa_v1.1.h2  hpa_v1.1.h3  sample_data
```

Conclusions

Colab is a fantastic tool but crashes frequently which caused many days of lost work. It is also limited in available RAM, which prohibited being able to have a reasonably competitive competition score. Colab documentation says there is 12 GB RAM available, but this is misleading as the 12 GB is shared amongst users. Through much trial and error it was determined to have a limit of about ~500 MB. There is not much online user knowledge in this area.

The problem was reasonably complex and required much work in understanding how to setup the problem. For instance, which images to train on. The competition stated that predictions were to be made on the green filters and that the other filters were for reference, which was a misleading statement as stacking the images provided better training scores. As another example, gaining an understanding on how to measure success, which required a custom metric.

Even though I was limited by the number and resolution of the training images, I believe my score could be improved further. As image sparcity was a major problem, if time had permitted, I would have tried to 'cherry pick' images from the full dataset in order to more fully represent the classes, by making sure I selected all the available images from the sparse classes in the full dataset volume.

For the classes where I had a good number of training images available, the network classified well.

In all, this was a fun problem and a great learning experience. Next steps are to take this learned knowledge and improve the results...!