Kaggle Diabetic Retinopathy Detection competition report

Ben Graham*
b.graham@warwick.ac.uk

August 6, 2015

1 Overview

The KaggleDiabetic Retinopathy Detection competition ran from February to July 2015. I competed under the team name Min-Pooling, and achieved a final Kappa score of 0.84958.

I preprocessed the images using Python and OpenCV to compensate for different lighting conditions. I then classified the images using SparseConvNet[1] convolutional neural networks (CNNs) running on an NVIDIA GPU. Finally, I used Python/Scikit-Learn to train a random forest to combine predictions from the two eyes into a single prediction, and output the final submission file.

*Department of Statistics and Centre for Complexity Science, University of Warwick, Coventry, UK
[1] https://github.com/btgraham/SparseConvNet

2 Software

To evaluate photographs of the retina using this method, access the "kaggle_Diabetic_Retinopathy_competition" branch of the SparseConvNet (GPL v3) GitHub repository:

`github.com/btgraham/SparseConvNet/tree/kaggle_Diabetic_Retinopathy_competition`

- The files kaggleDiabetes{1-3}.cpp contain the CNN configuration.
- The directory Data/kaggleDiabeticRetinopathy/ contains scripts for image preprocessing and model-averaging.
- Weight for trained networks can be made available via Dropbox.

Just in case anyone feels like taking a picture of their eyes with a smartphone, and then using some software they just downloaded from the internet to self-diagnose, I will point out that the GPL contains a disclaimer! See Figure 1.

Figure 1: Please note Section 15 of the GPL license: THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

3 Cohen's Kappa

The competition was scored using Cohen's quadratically weighted Kappa function, which takes integer predictions to produce a measure of similarity between the submitted ratings and the expert ratings. Averaging the output of the random forests produces a floating-point valued rating for each eye between 0 and 4. These need to be rounded to give an integer. The quadratic weights introduce an element of optimization into the rounding process: if an eye is probably a zero, but could be a 1 or a 2, you may be better off predicting 1 to avoid the possibility of getting a penalty of 2^2. In practice, you just have to determine "threshold" values to decide where to switch from rounding down to rounding up. See Figure 2.

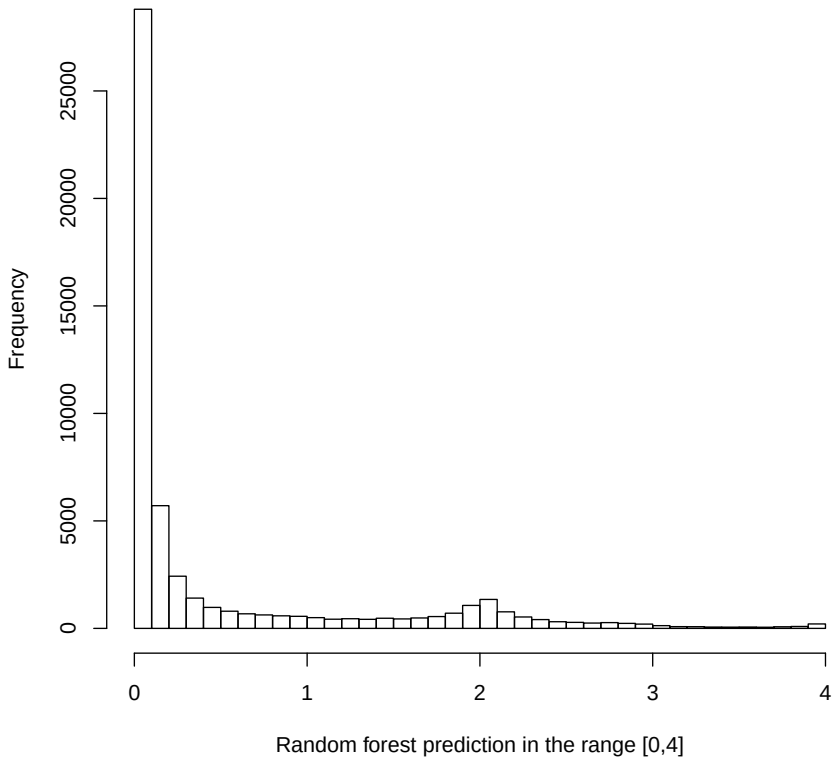

Figure 2: Distribution of predictions produced using random forests to combine left/right eye predictions. These predictions are divided in the classes 0,1,2,3,4 using the threshold values $\{0.57, 1.37, 2.30, 3.12\}$.

As I am not very familiar with Kappa scores, I was curious as to how much the private leaderboard would differ from the public leaderboard. See Figure 3. β is almost exactly one, so there is no regression towards the mean. Instead, final leaderboard scores are fairly uniformly a little lower than the public ones.

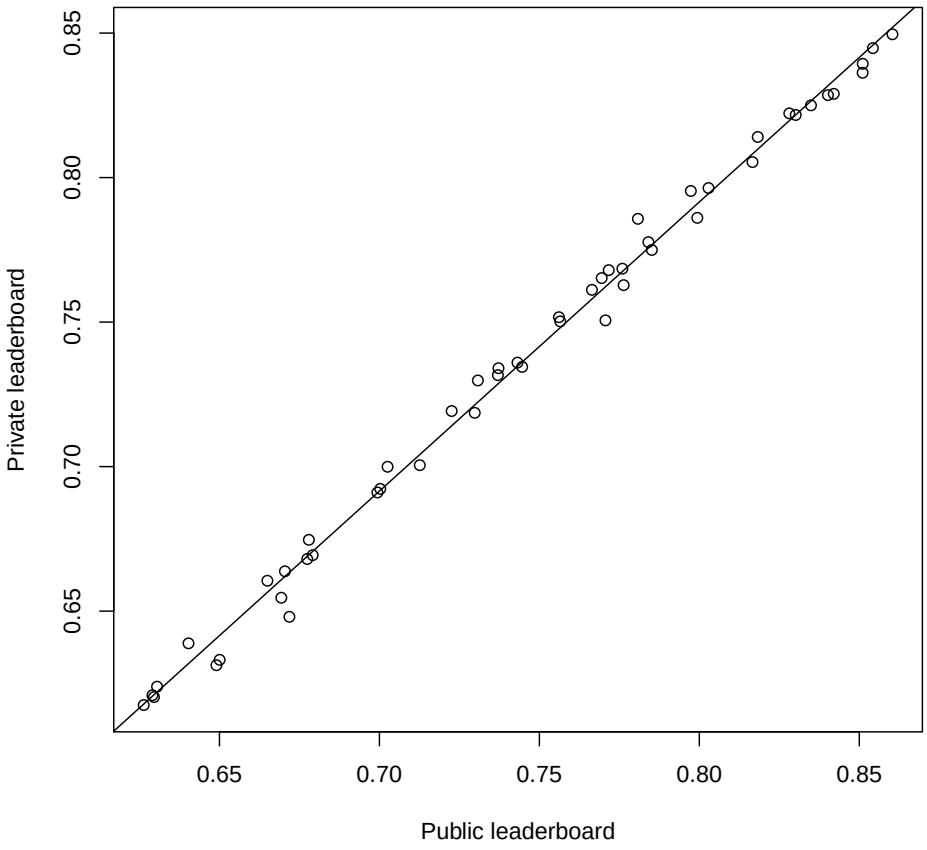

Figure 3: Comparison of public/private weighted Kappa scores for teams finishing in public leaderboard top-50. Fitting a linear regression model ("line of best fit") gives

$$\text{Private} = \alpha + \beta \times \text{Public} + N(0, 0.005257^2),$$

$$\alpha = -0.008633(\pm 0.007952), \qquad \beta = 1.00239(\pm 0.010628).$$

4 Image preprocessing

I preprocessed the images using OpenCV to

1. rescale the images to have the same radius (300 pixels or 500 pixels),
2. subtracted the local average color; the local average gets mapped to 50% gray,
3. clipped the images to 90% size to remove the "boundary effects".

This was intended to remove some of the variation between images due to differing lighting conditions, camera resolution, etc. Here are two before/after examples.

Image: 13_left
Rating: 0

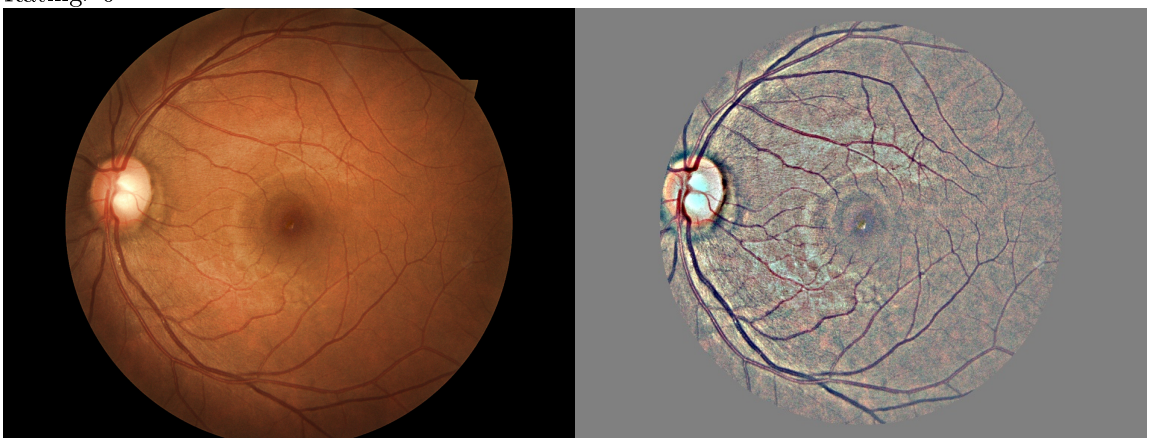

Image: 16_left
Rating: 4

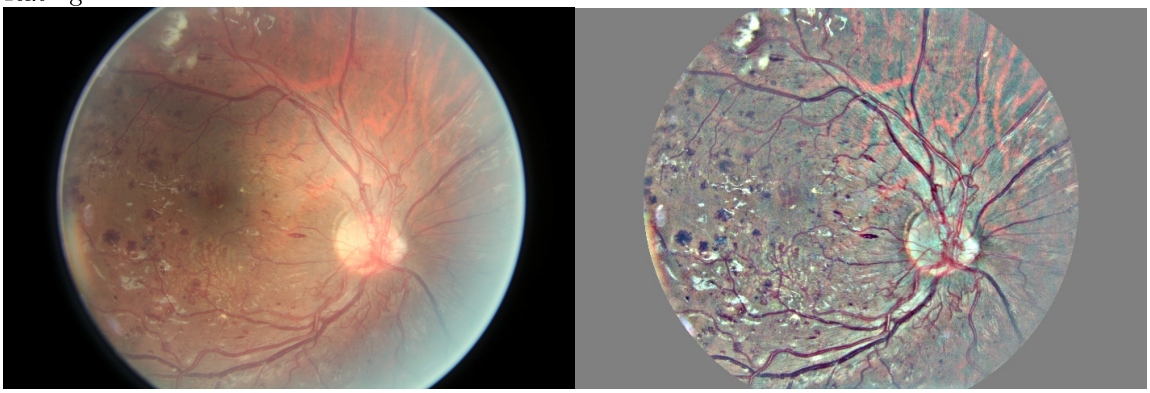

Figure 4: Two images from the training set. Original images on the left and preprocessed images on the right.

Algorithm 1 OpenCV Python code to preprocess the images.

```
import cv2, glob, numpy
def scaleRadius(img,scale):
    x=img[img.shape[0]/2,:,:].sum(1)
    r=(x>x.mean()/10).sum()/2
    s=scale*1.0/r
    return cv2.resize(img,(0,0),fx=s,fy=s)

scale=300
for f in glob.glob("train/*.jpeg")+glob.glob("test/*.jpeg"):
   try:
        a=cv2.imread(f)
        #scale img to a given radius
        a=scaleRadius(a,scale)
        #subtract local mean color
        a=cv2.addWeighted(a,                                    4,
                          cv2.GaussianBlur(a,(0,0),scale/30),-4,
                          128)
        #remove outer 10%
        b=numpy.zeros(a.shape)
        cv2.circle(b,(a.shape[1]/2,a.shape[0]/2),
                   int(scale*0.9),(1,1,1),-1,8,0)
        a=a*b+128*(1-b)
        cv2.imwrite(str(scale)+"_"+f,a)
    except:
        print f
```

5 Data augmentation

SparseConvNet can augment training and test images as the batches are formed, using OpenCV's warpAffine function. For training, the images are

1. randomly scaled by $\pm 10\%$
2. randomly rotated by between 0° and 360° degrees,
3. randomly skewed by ± 0.2.

For testing, the images are just rotated randomly.

I tried adding color-space transformation to the data-augmentation space. However, this did not seem to help. Perhaps because the images are already preprocessed to have zero-mean in any local region.

6 Network configurations

Given a single test image, you can attempt to classify it several times:

- using the same network, but randomly rotating the image, and/or
- using independently trained networks.

The results of the multiple tests are then averaged to give a single prediction. This kind of ensembling often yields substantial improvements in terms of accuracy, albeit with an increase in the complexity and computational cost. Even adding fairly weak networks to an ensemble can improve performance [2].

I trained a number of models and evaluated them on a (quite small) validation set of 1000 images. Combining the models seemed to offer very limited benefit. I decided to form an ensemble of the three best models. This seemed to perform only marginally better than the individual models. Following on from the competition, I hope to train a single model with comparable performance.

The three models consisted of two convolutional networks using fractional max-pooling, [3] and one based on the work of Simonyan and Zisserman[4]. The networks use 5-class softmax to predict a class 0,1,2,3,4. The output probabilities are to be used as the input for a random forest.

`github.com/btgraham/SparseConvNet/blob/kaggle_Diabetic_Retinopathy_competition/kaggleDiabetes1.cpp`
`github.com/btgraham/SparseConvNet/blob/kaggle_Diabetic_Retinopathy_competition/kaggleDiabetes2.cpp`
`github.com/btgraham/SparseConvNet/blob/kaggle_Diabetic_Retinopathy_competition/kaggleDiabetes3.cpp`

[2] See the ILVSRC and Kaggle National Data Science Bowl competitions for ample evidence of this.

[3] Fractional max-pooling `http://arxiv.org/abs/1412.6071`

[4] K. Simonyan, A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition

Radius	*:=Pooling type	Architecture	Complexity
270	Pseudorandom fractional max pooling 3x3 $\alpha = 1.8$	32C5-*-64C3-*-96C3-*-128C3-*-160C3-*-192C3-*-224C3-*-256C3-*-228C3-*-320C2-352C1-5N	2.7 million weights 5 GigaMultiplyAdds/image
270	Pseudorandom fractional max pooling 3x3 $\alpha = 1.5$	32C5-*-64C3-*-96C3-*-128C3-*-160C3-*-192C3-*-224C3-*-256C3-*-228C3-*-320C3-*-352C3-*-384C3-*-416C2-448C1-5N	6.1 million weights 14 GigaMultiplyAdds/image
450	Max pooling 3x3 stride 2	32C3-32C3-*-64C3-64C3-*-96C3-96C3-*-128C3-128C3-*-160C3-160C3-*-192C3-192C3-*224C3-224C3-*-288C2-288C2-320C1-5N	3.0 million weights 26 GigaMultiplyAdds/image

- For the first two networks, I used approximately 10% dropout in the final four layers. Increasing the number of feature vectors and using more dropout did not seem to help.
- Increasing the scale beyond 270 slowed down the processing without any apparent benefit.

7 Predictions: From softmax to integer scores

For each image in the training and test sets, I averaged the probabilities over the different networks (and repeat tests) to give a single probability distribution. To create a larger feature vector, I appended some meta-data

- the probability distribution for the person's other eye (left↔right),
- the size of the original images,
- the variance of the original images,
- the variance of the preprocessed images.

I then trained a random forest on the training images using scikits-learn. Classifying the test images with the predict_proba function gave each test image a score in the range $[0, 4]$. Thresholding then gave the final scores.

Using a random forest in this way is potentially dangerous. Ideally, the training data and test data for a random forest should be homogeneous. This is not the case as for the training images, we are using CNNs trained on those images, but with the test set we are using the same CNN, for which the test images are unseen data. Luckily, there does not seem to be much overfitting, so the approach worked.

Using a random forest here is probably over-kill. Using linear regression on just the CNN-generated probability distributions, without any of the meta-data, also seems to work well.

Acknowledgements

Thank you very much to the competition organizers at Kaggle, to EyePACS for providing the competition data and to the California Healthcare Foundation for sponsoring the competition.

Many thanks to everyone who discussed the competition on the Forum.
Finally, thanks to OpenCV and scikit-learn for making such useful software.