Diabetic Retinopathy Detection Using Convolutional Neural Networks

Zachary Daniels
Rutgers University
New Brunswick, New Jersey
zad7@cs.rutgers.edu

Kevin Soucy
Rutgers University
New Brunswick, New Jersey
kevin.soucy@rutgers.edu

Sachin Srivastava
Rutgers University
New Brunswick, New Jersey
sachin.srivastava@rutgers.edu

Abstract

Diabetic retinopathy is a disease where the retina of the eye is damaged as a result of diabetes and if left untreated, can eventually lead to blindness. Using data made available by the California Healthcare Foundation and EyePACS as part of a competition hosted by Kaggle [6], we explore diabetic retinoplasty detection using techniques borrowed from computer vision, signal processing, pattern recognition, and machine learning. Specifically, we apply convolutional neural networks to first classify retinal scans as healthy or unhealthy and to then predict the stage of the disease on the unhealthy images.

1. Introduction and Problem Statement

Diabetic retinopathy (DR) is a disease of the eye where the blood vessels of the retina, the light-sensitive area located in the back of the eye, are damaged from diabetes, a metabolic disease characterized by the body's failure to produce or properly interact with insulin [15] [13]. If left untreated, diabetic retinopathy can lead to blindness. In fact, diabetic retinopathy is the number one cause of blindness in the working-age population in developed countries, currently affecting over 93 million people [6]. The current method for detecting diabetic retinopathy requires specially trained clinicians carefully examining fundus photographs of the retina, a very time-intensive, human-centric process. As a result, the California Healthcare Foundation has sponsored a Kaggle competition to automate this process, and we have elected to participate in this competition [6].

The goal of the Kaggle competition is to take a high-resolution image of the retina as input and output a predicted score based on how a trained clinician would rate the severity of the disease. The severity is rated as belonging to one of the following classes: "No", "Mild", "Moderate", "Severe", or "Proliferative" DR.

2. Literature Review

A great deal of work has been done on the topic of automatically analyzing digital images of eyes for diabetic retinopathy. Walter *et al*. focused on detecting retinal exudates, fluids emitted within the retina, as a first step for the diagnosis of DR by looking at variations in gray-level, finding contours using morphological operators, and applying a segmentation algorithm [16]. Exudate detection was a focus of Zhang *et al*. as well. Similar to Walter, they used a morphological segmentation algorithm but performed more complex pre-processing by normalizing and denoising the image while also detecting image artifacts and reflections. A random forest algorithm was used for classification [18].

Franklin and Rajan were another team to explore exudate detection [2]. Their contribution was to use the Luv colorspace and apply feature extraction techniques borrowed from image processing. A neural network was used for binary classification on a per-pixel level. Like Franklin, Gardner *et al*. used an artificial neural network to train a classifier, but instead of only focusing on exudates, they tried to predict whether image patches represented normal eye tissue, blood vessels, exudates, or hemorrhages [4].

Sinthanayothin *et al*. focused on detecting DR before it lead to full blindness. They pre-processed color images of the eye, identified landmark points (namely, the optic disc, blood vessels, and the fovea), and then performed segmentation of lesions related to DR [14]. While Gardner focused on detecting non-proliferative DR, Welikala *et al*. focused on the opposite case. One key feature for proliferative DR is neovascularization, "the growth of abnormal new vessels" [17]. Welikala used a line operator to detect lines in the image and segment blood vessels. They extracted features along these vessels and used an SVM to classify vessels as "pre-existing" or "new" [17].

Ganesan *et al*. extracted features using trace transforms and then applied SVMs and probabilistic neural networks for detecting early stage DR [3]. Trace transforms are generalizations of the radon transform, and the mathematical details can be found in [3]. Antal and Hajdu examined combining six existing automated DR screening systems and classifiers into one system using an ensemble of classifiers [1]. A survey on other existing techniques and systems for various stages of the DR detection pipeline was authored by Mookiah *et al*. [11].

3. Methodology

Our system is composed of three stages and is illustrated in Fig. 1. In the first stage, we preprocess our image data using standard techniques from image processing. In the second stage, we attempt to classify eye images into two general classes: healthy (no DR) vs unhealthy (mild, moderate, severe, and proliferative DR). In the final stage, we try to predict the specific severity of DR in unhealthy eyes using regression.

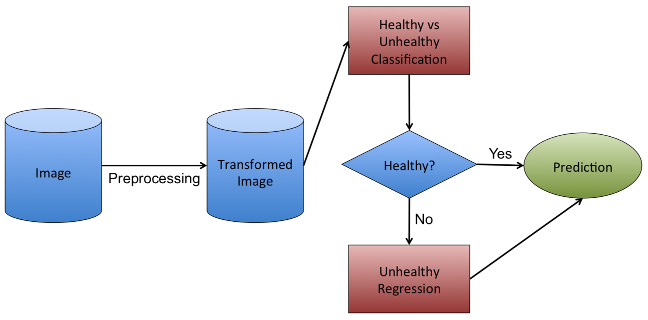

Figure 1: Our system architecture

3.1. Technical Details

Our system is written mostly in Python with some additional code written in MATLAB. We rely heavily on external libraries and software including skimage, Berkeley's Caffe deep learning framework, Gimp, and imagemagick. Our solution is hosted on Amazon's EC2 platform.

3.2. Dataset

The dataset consists of a large set of high-resolution retina images provided by the California Healthcare Foundation and EyePACS. There are several inconsistencies among the images that must be controlled. These include differences in lighting and differences in the model and type of camera. According to the California Healthcare Foundation, images may contain artifacts, be out of focus, be underexposed, be overexposed, and/or be taken from one of two viewpoints: anatomically or inverted (what one sees during a typical live eye exam). A left and right field is provided for every subject.

Our training dataset consists of 35,127 images of various size, quality, orientation, eye positioning, and exposure, totaling 45GB of data. The test/validation dataset consists of 53,577 images, totaling 60GB of data. The number of training images belonging to each severity level is shown in Table 1. Note the data is highly skewed. To address this problem, we experimented with both downsampling to match the minority label and upsampling to match the majority label.

Severity	Number of Training Images
No DR	25,798
Mild DR	2,439
Moderate DR	5,286
Severe DR	873
Proliferative DR	707

Table 1: The split of the dataset by severity of DR

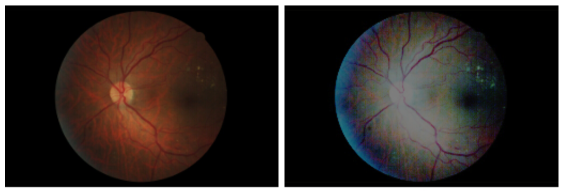

Figure 2: Example of a standard image in the dataset (left) and an equalized version (right)

3.3. Data Preprocessing

Image preprocessing is required as the images may have differing image dimensions and image quality. We require a robust mechanism for unifying image sizes, centering the eyeballs, and dealing with other miscellaneous noise and variance inherent in the image classification problem (*e.g.* differences in illumination and contrast). We explore the following preprocessing methods:

- Histogram equalization on intensity images and on each color channel in RGB space: There are significant differences in lighting and coloration between the images in our dataset. In an attempt to overcome the effects of such differences, we first try converting the image to a grayscale intensity image and then use histogram equalization to adjust the contrast of the image. We find that removing color information negatively affects the performance of our model. We then try applying histogram equalization to each of the three RGB color channels and recombining them. This has the desirable effect of making the already distinct veins and hard exudates sharply contrast with the rest of the eyeball, but it also reduces the contrast between some of the soft exudates, some of the less distinct veins, and the gel of the eyeball which lowers classification accuracy between health and unhealthy eyes. An example of the original image and the equalized image is shown in Fig. 2.

- Centering and cropping the eyeball: After converting the image to grayscale, we use a threshold to segment the background of the image from the foreground (the eyeball). We then crop the image to the borders of the eyeball.
- Image resizing: We resize all images to 256 pixels x 256 pixels. When the aspect ratio of the original image differs from 1:1, we add borders to maintain the original aspect ratio and keep the eyeball centered.
- Mean image subtraction: To build our model, we take an existing, pretrained convolutional neural net, and finetune it on our data. This is explained in greater detail in Section 3.4.1. In order to center the pixels of the images from the DR dataset, we subtract the mean image of the data used to train the original net from each image in the DR dataset.

We use the Python version of the skimage library, the photo-editing program Gimp, and the imagemagick software suite for most of the image processing-related functionality (script modified from [7]).

3.4. Feature Extraction and Classification

We use convolutional neural networks (CNN) for learning and extracting features as well as for performing classification and regression. We use the Caffe library with Python bindings for working with CNNs [5].

3.4.1 Convolutional Neural Networks: An Overview

CNNs have been proven to be very effective tools for recognition- and detection-related tasks in computer vision. We decided to apply CNNs to our problem for two reasons. First, CNNs tend to scale well with large amounts of data. Second, CNNs are able to do end-to-end feature learning, feature extraction, and classification and/or regression, removing the need to hand-engineer features.

Convolutions neural nets first appeared in the paper by LeCun *et al*. [9]. CNNs are loosely modeled on how the brain works, specifically mimicking the visual system. In the first layer, CNNs tend to learn filters which are typically directional bars which capture edges at specific orientations and spots of colors. These filters correspond to weights in the neural network. The filters learned for our regression task are shown in Fig. 3. These filters are used to extract features for classification and regression tasks. CNNs are a “deep” architecture: they learn hierarchies of features. For example, a CNN designed for recognizing faces might learn edges in its first layer, group these edges into components such as mouths, noses, and eyes in its second layer, and learn full face templates in its third layer.

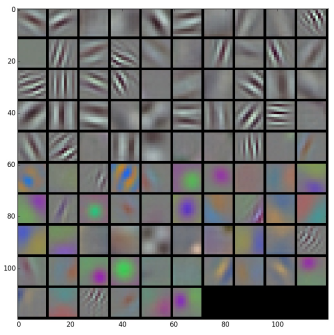

Figure 3: Filters learned on the regression net

CNNs are trained using backpropagation. First, a forward pass through the network is performed which results in a set of predictions (either classifications or reconstructions). The error for these predictions are computed based on some loss function and the error is backpropagated through the network: weights are adjusted in order to minimize the error. This process is repeated for some specified number of iterations.

CNNs do have limitations: they form very complex decision boundaries, and if they have a large number of neurons, they have a very large capacity (ability to memorize the data) making them especially prone to overfitting. As such, they require careful design decisions and tuning of a (typically) very large number of hyper-parameters. Some of the issues we have to deal with include:

- Selecting/designing the network architecture (*e.g.* depth of the net, the size and types of layers, size and number of filters, pooling method, activation layer type, loss type, output layer classifier type, etc.)
- Selecting and tuning the optimization method and parameters used for backpropagation
- Reducing overfitting
- Dealing with issues related to computationally heavy tasks (*e.g.* using the GPU for training)

3.4.2 The AlexNet Architecture

AlexNet (pictured in Fig. 4) is the among the fist CNN architectures that performed well on the task of general object recognition [8]. It consists of 650,000 neurons with 60 million total hyperparameters. CNNs generally have three major types of layers: the convolutional layers, pooling layers, and fully-connected layers. The general relationships between these layers are pictured in Fig. 5. Convolutional layers perform the feature extraction. Pooling layers downsample the convolutional layers (in our case using max-pooling), significantly reducing the total number of hyperparameters. The fully-connected layers are responsible for

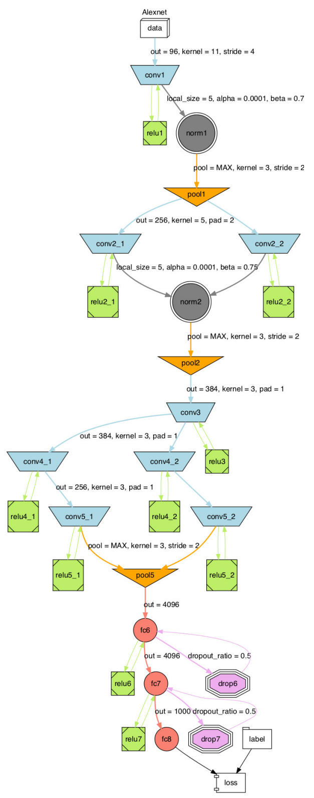
Alexnet
data
out = 96, kernel = 11, stride = 4
conv1
local_size = 5, alpha = 0.0001, beta = 0.7
relu1
norm1
pool = MAX, kernel = 3, stride = 2
pool1
out = 256, kernel = 5, pad = 2
conv2_1
conv2_2
local_size = 5, alpha = 0.0001, beta = 0.75
relu2_1
norm2
relu2_2
pool = MAX, kernel = 3, stride = 2
pool2
out = 384, kernel = 3, pad = 1
conv3
out = 384, kernel = 3, pad = 1
conv4_1
conv4_2
relu3
out = 256, kernel = 3, pad = 1
relu4_1
conv5_1
relu4_2
conv5_2
pool = MAX, kernel = 3, stride = 2
relu5_1
pool5
relu5_2
out = 4096
fc6
out = 4096
dropout_ratio = 0.5
relu6
fc7
drop6
out = 1000
dropout_ratio = 0.5
relu7
fc8
drop7
label
loss

Figure 4: The AlexNet CNN architecture, source: [12]

making predictions. AlexNet is composed of five convolutional layers, three max-pooling layers, and three fully-connected layers. AlexNet starts with kernels (filters) of size 11x11 in the first layer and ultimately end up with kernels of size 3x3 in the final convolutional layer. Another important aspect of CNNs is the type of activation function used. AlexNet uses rectified linear units which are very fast to compute when compared to the tanh and sigmoid functions. Lastly, AlexNet uses Dropout in its first two fully connected layers to prevent overfitting. Dropout works by zeroing out neurons' weights with some specified probability (0.5 in our case). We experiment with modifying the AlexNet architecture, but because we decided to use transfer learning instead of training the network from scratch (explained in detail in the following section), we ultimately revert to using the standard formulation of AlexNet as our network architecture with some minor tweaks (excluding a completely changed final layer).

3.4.3 Transfer Learning

We originally tried to train the net from scratch, i.e. using random initial weights, but training from scratch has two major disadvantages: it takes a very long time to train (potentially a week or more), and it requires large amounts of data. As a result, we instead choose to apply the concept of transfer learning. Transfer learning is the process of taking a model constructed to address problems in one domain and applying it to problems in another. In neural nets, this means taking the filters (weights) learned on one dataset and using them to initialize the weights in another network and then finetuning the net on new data. We take a net trained on a subset of ImageNet, consisting of millions of images of 1000 general object classes, drop the last layer (the loss layer), replace it with our own, and retrain on images from our dataset. Compared to training from scratch, using transfer learning for CNNs requires far less data since the net has already been trained on large datasets, and the loss converges a lot faster, so training time is significantly reduced.

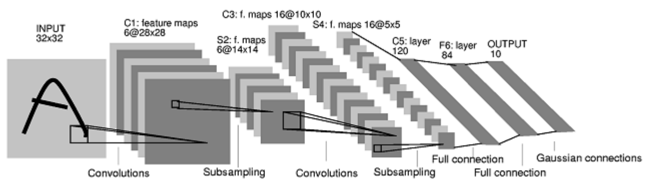

Figure 5: A broad overview of CNNs and the relationships between layers, source: [10]

3.4.4 Data Augmentation

CNNs have a tendency to severely overfit the training data. There are four general ways to overcome this problem: one can apply dropout to the later layers of the network, one can add a regularizer to the loss function, one can perform early stopping (stop training when the validation metric plateaus), and one can attempt to find more data to train the net on. We significantly augment the data in an attempt to prevent overfitting. We augment our data in the following ways:

- Random mirroring at training time
- Random cropping (and scaling) at training time
- Rotating all images by 90, 180, and 270 degrees
- Adjusting the contrast, hue, and saturation of all images

Ultimately, we generate 24 times the amount of initial data (excluding the transformations at training time).

3.4.5 Parameter Tuning for CNNs

CNNs have large numbers of parameters that must be tuned. Some of the major parameters that need to be tuned include:

- Loss function: the function that the network is trying to optimize. Caffe supports three types of loss functions out of the box: the softmax and hinge losses for classification and the Euclidean loss for regression. After experimentation, we decided to use the hinge loss for our classification net and the Euclidean loss for our regression net.

- Optimization method: determines how well the loss converges and the speed of the convergence. Caffe supports three types of optimization methods: stochastic gradient descent (with or without momentum), Nesterov's accelerated gradient descent, and AdaGrad. For our problem, AdaGrad outperformed the other methods.
- Learning rate: determines the step size during optimization. Our learning rate started at 0.01.
- Decay rate and decay step size: The decay rate dictates how much the learning rate should decay after every specified number of iterations. We decayed the learning rate by a factor of 0.1 every 30,000 iterations for the classification net and every 10,000 iterations for the regression net.
- Momentum: helps reduce the energy of the system during optimization in an attempt to keep the parameters from zigzagging around too much. If set too low, the loss bounces around too much, and if set too high, the loss instantly converges and gets stuck in a local minimum. Switching to AdaGrad, we no longer have to worry about momentum, and we expect this is why it worked well for our problem. Alternatively, we could have tried starting with a low momentum and slowly increasing it over time.
- Batch size: the number of examples to use for each iteration of the learning process. A larger batch size results in a smoother loss decay, but requires more training time and is more memory intensive. Our batch size was 250 images per batch.

3.4.6 Threshold Determination for Regression

Our second net outputs continuous values, but the problem requires discrete classes as outputs because the problem is actually an ordinal regression problem. The simplest thing to do is round the outputs to the nearest integer bounded below by one and above by four. There is no guarantee or reason to believe that the severity of DR is linear between each discrete category. We experimented with selecting different thresholds for converting the continuous outputs to discrete ones. We used a genetic algorithm to attempt to minimize the sum of the (1) absolute and (2) squared differences between predictions and true class values for the entire training set at different thresholds. Adjusting the thresholds did not prove to be too helpful: we saw very minor improvements on the test data, and the threshold values ended up being very close to the midpoints of the ordinal values. In retrospect, we could have easily applied a brute force approach to selecting the threshold instead of using genetic algorithm optimization since we only have three thresholds each of which has a range of one.

3.4.7 Ensemble of Models

One common way to improve performance on a problem is to combine models in an ensemble. There are several ways to ensemble CNNs:

- Use CNNs trained with different weight initializations
- Use CNNs trained on different subsets of the data
- USe CNNs trained using different parameters
- Train a single CNN, but take "snapshots" of it at different stages in the learning process and use these snapshots
- Extract features from a single CNN and train a number of different types of classifiers on it

We took snapshots of our classification net every 2000 iterations and of our regression net every 1000 iterations. While the nets with more iterations generally have better performance than those with fewer, performance on the loss tends to decay in an almost exponential manner, so later nets will still have similar performance even if there are minor performance differences. We run 6 classification nets and pipe the output of each of these nets into 6 regression nets. We build an ensemble of 36 nets. We also experiment with only using the top 3 nets from both regression and classifications, for an ensemble size of 9 nets. For each example in the test set, we experiment with taking majority vote, maximum value, and the mean of the predicted class labels. We find that our method of constructing ensembles doesn't help as much as simply adjusting the thresholds for rounding.

3.5. Experiments and Evaluation

3.5.1 Evaluation Metrics

Each training image is labeled by a pair of graders with score i belonging to a human grader and j belonging to an automated grader. These scores fall in $\{0, 1, 2, 3, 4\}$. The difference in scores is calculated as

$$w_{i,j} = \frac{(i-j)^2}{(N-1)^2}$$

where N is the total number of possible combinations of scores. Histograms of the number of witnessed score combinations O and expected ratings for each score combination E are computed, and the evaluation criteria, the quadratic weighted kappa, is computed as:

$$\kappa = 1 - \frac{\sum_{i,j} w_{i,j} O_{i,j}}{\sum_{i,j} w_{i,j} E_{i,j}}$$

We also use other common metrics including the accuracy rate and mean squared error.

3.5.2 Experiments

We trained over twenty nets, but instead of describing each net trained, we will briefly focus on six broad categories of experiments and what we learned from them. We will ignore our very brief experimentation with training a net from scratch since we quickly realized this was not the best path to take and most of our effort was focused on finetuning a preexisting net.

Experiment 1: finetuning, intensity images, unbalanced data

Our first set of experiments involved converting our images to intensity images and equalizing the image. Our first classification net (which predicted each individual DR score class) achieved an accuracy rate of 73%. The net nearly instantly converged. There are two reasons for this. First, we didn't yet know how to properly adjust the parameters. Second, 73% of the data belonged to the "No DR" class, and it soon became clear that the net was assigning all the data to the majority class.

Experiment 2: finetuning, intensity images, downsampling

To fix the unbalanced data problem, we tried downsampling the data so the number of healthy examples matched the number of unhealthy examples. We also decided to do binary classification (healthy vs unhealthy) and then perform regression on the predicted unhealthy instances. We saw a drop in accuracy, but we were no longer predicting that every instance belonged to the majority class.

Experiment 3: finetuning, color images, downsampling

We then tried training on the raw color images instead of the intensity images and saw minor improvements in accuracy in the classification task.

Experiment 4: finetuning, color images, upsampling

Convolutional neural networks require a large amount of data in order to prevent or delay overfitting. When we did downsampling, we lost a large amount of potentially useful data. Instead, we tried upsampling the minority class by repeating images. Since we had a relatively small batch size compared to the total number of images, within batch repetition wasn't a major issue. At this point, we were achieving around 67% accuracy on the classification task.

Experiment 5: finetuning, color images, upsampling, data augmentation

Another way we attempted to prevent overfitting was by augmenting the dataset as previously described. This helped fairly significantly, we achieved around a 70% accuracy on the classification task and a 0.82 mean squared error on the regression task. Likewise, training on 80% of training data resulted in a quadratic weighted kappa score of around 0.25 on the holdout test set.

Experiment 6: finetuning, color images, upsampling, data augmentation, trained on all data

Finally, we trained on the full data, and trained for more iterations (42,000 for classification, 30,000 for regression). We achieved an accuracy rate of 88% on the classification task and a mean squared error of around 0.6 on the regression task on the validation data. However, it should be noted that these are overly biased estimates of the evaluation metrics because in this one case, the training data included the validation data, so we were training and testing on the same data. We also experimented with automatically adjusting the rounding thresholds and ensembling snapshots of our CNN with limited success. We were able to improve our weighted kappa score on the holdout test data to 0.36753, a significant improvement from training on only 80% of the training data.

3.6. Final Results

For our final model trained on the training data and evaluated on a disjoint validation set, we achieved an accuracy rate of 0.7 for the healthy vs unhealthy classification task and a mean squared error of 0.82 for the regression on DR severity task. If we compare loss on the training set for the classification task (Fig. 6) to the accuracy on the validation set for the same task (Fig. 7), we see an inverse trend. As loss decays, we expect accuracy to grow. Typically with CNNs, we see the validation accuracy plateau long before the training loss plateaus, and this is indicative of overfitting. For example, Fig. 8 shows that when the data is unaugmented, we can memorize the training data, driving the loss to zero. If we were to also plot the validation accuracy, one would see that the accuracy plateaus many iterations earlier meaning the net is likely severely overfitting by 40,000 iterations.

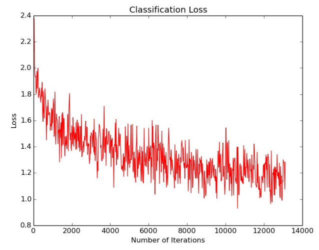

Figure 6: Hinge loss on the training data on the healthy vs unhealthy classification task

On the regression task, we're typically off by about one degree of severity. If we look at the decay in the training loss and validation loss (pictured in Figs. 9 and 10 where loss is one-half the mean squared error), we see that they both quickly drop, and then the training loss decays very slowly while the validation loss quickly plateaus. This is likely due to the amount of available training data being much smaller than in the classification task.

We also examine the errors being made. On the classifica-

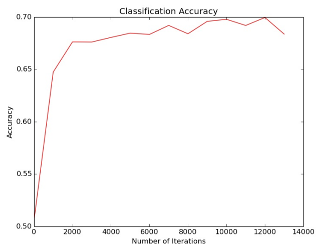

Figure 7: Accuracy on the validation data on the healthy vs unhealthy classification task

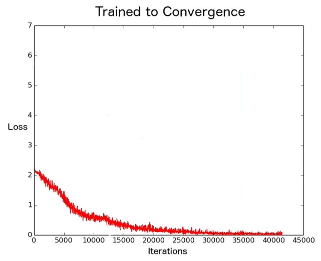

Figure 8: Given enough time and without using data augmentation, loss on training data will converge to zero, *i.e*. the CNN has memorized the training data

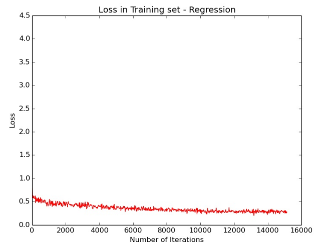

Figure 9: One-half the mean squared error on the training data on the regression task

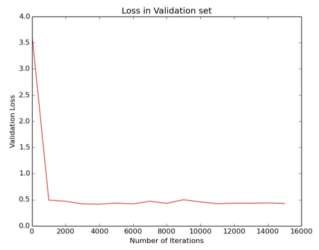

Figure 10: One-half the mean squared error on the validation data on the regression task

tion task (Fig. 11), most confusion occurs between the DR severities of 0 (No DR) and 1 (Mild DR). Taking this into consideration and looking at what other Kaggle competitors have done, it might have been better to train the initial classifier on {0,1} vs {2,3,4} instead of healthy vs unhealthy.

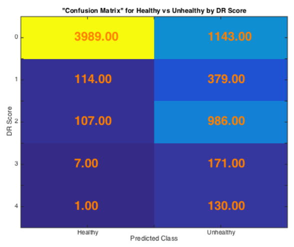

Figure 11: The “confusion matric” for the classification task

Looking at the regression task (shown in Fig. 12), we see that instances tend to be grouped into {2,3} instead of {1,2,3,4}. Adjusting the rounding threshold helps to alleviate this problem, but to improve the regression further would probably require significantly more training data or building models for different subsets of the dataset (similar to one-vs-one or one-vs-all classifiers).

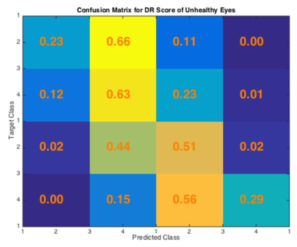

Figure 12: The confusion matrix for the regression task (with rounding)

When training on the full training dataset and testing on the validation set, we achieve an accuracy rate on the classification task of 88% and a mean squared error of around 0.6. Using these same nets with modified thresholds for rounding the regression outputs, we achieve a weighted quadratic kappa of 0.36753. This places our team (DRE) at 56 of 342 on the Kaggle leader board.

4. Extensions

There are a number of ways we might be able to improve our model:

- Experiment with alternative ways of building ensembles
- Directly perform ordinal regression by either replacing the loss layer in the network or extracting features using the CNN and then passing it to an ordered probit or logit model
- Perform automatic hyperparameter tuning using something like Bayesian hyperparameter optimization
- Use a different or additional data augmentation scheme
- Explore alternative, more modern CNN architectures like VGG, GoogLeNet, and Network-in-Network

5. Conclusions

We apply convolutional neural networks to the problem of detecting and rating the severity of diabetic retinopathy in digital images of eyes. While trying to solve this problem, we faced and continue to face several issues. We have to handle a very large amount of data. We have to deal with time constraints compounded by long training times. A lot of parameter tuning has to be done by hand. Disparities exist between the data used to learn the weights of the original neural net that we finetuned from and our own data. CNNs are very easy to overfit, and we still experience issues of overfitting even when using techniques such as dropout and large amounts of data augmentation. Lastly, the problem is difficult even for humans.

6. Distribution of Work

The distribution of work fell as follows:

- Kevin: Researching CNNs + DR, setting up AWS and installing Caffe and skimage with dependencies. Downloading/unzipping 100GB of datasets in ubuntu, running the first preprocessing scripts in python and gimp, finetuning AlexNet architecture, finetuning of solver parameters and training for first net (experiments 1-4), construction of unique downsampled and upsampled randomized training and validation sets for each experiment and updating file paths to processed images, worked on presentation and report.
- Sachin: Researching CNNs + DR, setting up Caffe, connecting Caffe to python. Building code for classification, regression of individual example images, database preparation, data visualization, result analytics: filter visualization + result graphs, worked on presentation and report
- Zach: Researching CNNs + DR, data preprocessing and augmentation, later tuning of solver parameters for classification net (experiments 4-6), tuning regression net parameters, minor tweaks to AlexNet, connecting classification + regression pipeline, experimented with automatic threshold determination and ensembling CNNs, worked on presentation and report

References

[1] B. Antal and A. Hajdu. An ensemble-based system for automatic screening of diabetic retinopathy. *Knowledge-Based Systems*, 60:20–27, 2014. 2

[2] S. W. Franklin and S. E. Rajan. Diagnosis of diabetic retinopathy by employing image processing technique to detect exudates in retinal images. *Image Processing, IET*, 8(10):601–609, 2014. 1

[3] K. Ganesan, R. J. Martis, U. R. Acharya, C. K. Chua, L. C. Min, E. Ng, and A. Laude. Computer-aided diabetic retinopathy detection using trace transforms on digital fundus images. *Medical & biological engineering & computing*, 52(8):663–672, 2014. 1, 2

[4] G. Gardner, D. Keating, T. Williamson, and A. Elliott. Automatic detection of diabetic retinopathy using an artificial neural network: a screening tool. *British journal of Ophthalmology*, 80(11):940–944, 1996. 1

[5] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 3

[6] Kaggle. Diabetic retinopathy detection. http://www.kaggle.com/c/diabetic-retinopathy-detection, Feb 2015. 1

[7] H. Koepke. Diabetic retinopathy preprocessing code. https://github.com/hoytak/diabetic-retinopathy-code/blob/master/prep_image.sh, March 2015. 3

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 3

[9] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989. 3

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 4

[11] M. R. K. Mookiah, U. R. Acharya, C. K. Chua, C. M. Lim, E. Ng, and A. Laude. Computer-aided diagnosis of diabetic retinopathy: A review. *Computers in Biology and Medicine*, 43(12):2136 – 2155, 2013. 2

[12] M. P. Nieto. Drawing of alexnet convolutional neural network. http://commons.wikimedia.org/wiki/File:Alexnet.svg, June 2014. 4

[13] W. H. Organization. About diabetes. http://www.who.int/diabetes/action_online/basics/en/. 1

[14] C. Sinthanayothin, J. Boyce, T. Williamson, H. Cook, E. Mensah, S. Lal, and D. Usher. Automated detection of diabetic retinopathy on digital fundus images. *Diabetic medicine*, 19(2):105–112, 2002. 1

[15] M. C. Staff. Diabetic retinopathy. http://www.mayoclinic.org/diseases-conditions/diabetic-retinopathy/basics/definition/con-20023311, Mar 2012. 1

[16] T. Walter, J.-C. Klein, P. Massin, and A. Erginay. A contribution of image processing to the diagnosis of diabetic retinopathy-detection of exudates in color fundus images of the human retina. *Medical Imaging, IEEE Transactions on*, 21(10):1236–1243, 2002. 1

[17] R. Welikala, J. Dehmeshki, A. Hoppe, V. Tah, S. Mann, T. H. Williamson, and S. Barman. Automated detection of proliferative diabetic retinopathy using a modified line operator and dual classification. *Computer methods and programs in biomedicine*, 114(3):247–261, 2014. 1

[18] X. Zhang, G. Thibault, E. Decencière, B. Marcotegui, B. Laÿ, R. Danno, G. Cazuguel, G. Quellec, M. Lamard, P. Massin, et al. Exudate detection in color retinal images for mass screening of diabetic retinopathy. *Medical image analysis*, 18(7):1026–1043, 2014. 1