ABSTRACT

Improving the Learning Platform for the Leukocoria Detection Project

James Boer, M.S.

Mentor: Gregory J. Hamerly, Ph.D.

In this paper we describe a new and significantly improved learning platform for the leukocoria detection project. We have developed an easily maintainable and extensible system that can train hundreds of semi-random convolutional neural networks very quickly on GPUs. Docker is used to enable simple and platform-agnostic deployment of the system. The Torch machine learning library is used to train the convolutional neural networks and a PostgreSQL database is used to store the training results. A literature review of recent convolutional neural network research is done to find methods of improving accuracy. Training results are promising, with single-network performance at 96.6% using minimal data and 99.5% using data augmentation.

CHAPTER ONE

Introduction

Retinoblastoma (Rb) is the most common ocular malignancy in children, occurring in 1 in 18,000 to 30,000 live births worldwide (Abramson and Schefler 2004). A child with Rb can develop one or more tumors in one or both eyes. Left untreated, retinoblastoma can advance towards the brain and cause death. Early detection allows treatments that can prevent death or surgical removal of the eye (Abramson and Schefler 2004).

The most common symptom of Rb is a white reflection emitted from the retina and seen through the pupil. The reflection of light off the tumor causes the white color. This symptom is called leukocoria and is present in 60% of reported Rb cases in the United States (Abramson and Schefler 2004). It has been concluded that the intensity of leukocoria is an indicator of the state of the malignancy (Abdolvahabi, Taylor, Holden, Shaw, Kentsis, Rodriguez-Galindo, Mukai, and Shaw 2013). An example of a normal eye can be seen in 1.1a compared to a leukocoric eye in 1.1b. Leukocoria does not always indicate Rb, as it can also indicate several other ocular diseases (e.g. Coats' disease and cataracts).

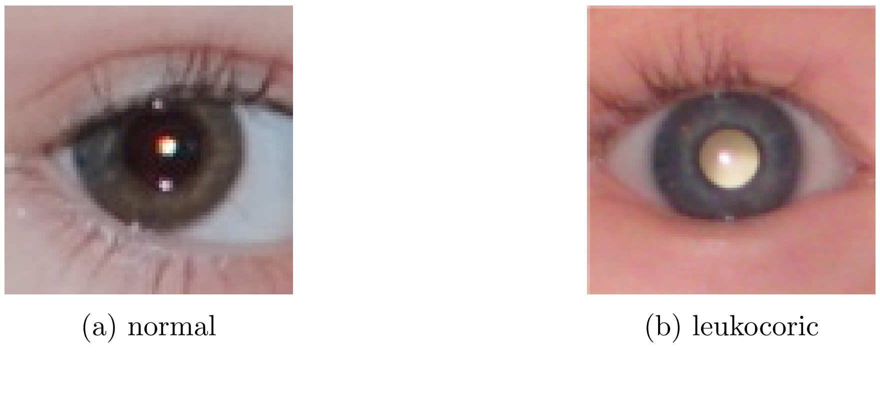

(a) normal (b) leukocoric

Figure 1.1. Examples of a normal and leukocoric eye. Credit: Ryan Henning et al. (2014)

Rb is most common in children age 5 or less, with most diagnoses occurring between 1 and 2 years of age (Ries, Smith, Gurney, Linet, Tamra, Young, Bunin, et al. 1999). While physicians screen for leukocoria, parents often detect it first (from pictures of their child), but are not aware of its connection to Rb. Therefore, automated leukocoria detection is valuable, as it increases the likelihood of earlier diagnosis of Rb or other ocular diseases. This offers motivation for the leukocoria detection project.

The goal of our project is to improve on the previous work done on leukocoria detection. This previous work and its known deficiencies are discussed in Chapter 3, which will provide motivation for our improvements. Ultimately, we develop an easily maintainable and extensible system that can train hundreds of semi-random convolutional neural networks very quickly on GPUs. Docker is used to enable simple and platform-agnostic deployment of the system. The Torch machine learning library is used to train the convolutional neural networks and a PostgreSQL database is used to store the training results. A literature review of recent convolutional neural network research is done to find methods of improving accuracy. Training results are promising, with single-network performance at 96.6% using minimal data and 99.5% using data augmentation.

CHAPTER TWO

Neural Networks and Convolutional Neural Networks

In this chapter we discuss neural networks and convolutional neural networks, and explain why convolutional neural networks are well-suited for the task of detecting leukocoria.

2.1 Neural Networks

Neural networks (NNs) are supervised machine learning models that are inspired by the biological connections in the brain. By training on some inputs and labels, the neural network will model some function that allows its outputs to match the given labels.

Traditional feed-forward neural networks are composed of multiple layers of neurons, where each neuron in a layer is connected to each neuron in the following layer by some learnable weight. Typically these weights are learned from the error backpropagation algorithm. The input to each neuron is composed of the dot product of the value and weight of its input neurons, to which some non-linearity is applied using an activation function like the hyperbolic tangent.

2.2 Convolutional Neural Networks

Traditional feed-forward neural networks are not well-suited for image classification. This is because neural networks treat all RGB values as independent. This ignores the significance of what an image is. When we look at images, we use the neighbors of pixels in order to determine where objects and features of objects begin and end. The idea behind convolutional neural networks (CNNs) is to change the data representation into something more useful for a fully-connected neural network to classify.

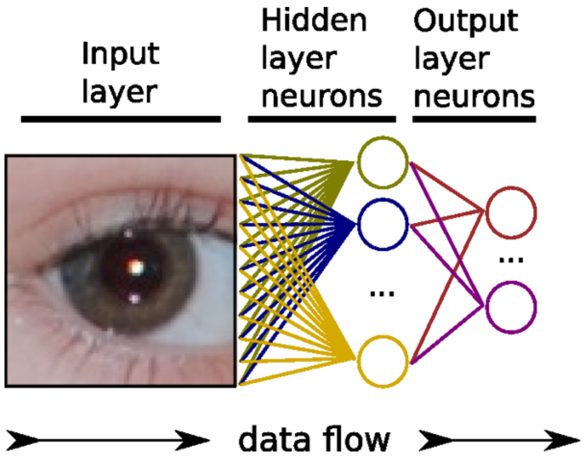


Figure 2.1. A fully-connected neural network. Credit: Ryan Henning et al. (2014)

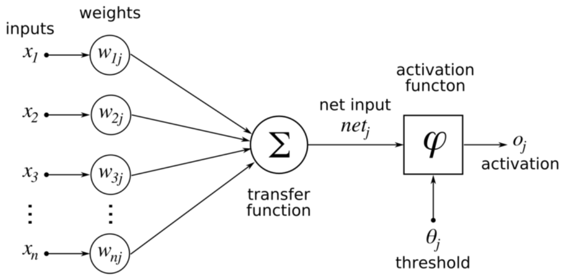


Figure 2.2. A single neuron in a neural network. Credit: Wikimedia

Convolutions are used to change the data representation. A convolution is simply a filter that “slides” over an image, computing the dot product of the filter weights and the image values. See Figure 2.3 for an illustration. In a CNN, an activation function is applied to each convolution result to provide non-linearity.

The filter weights in a convolution layer are learnable parameters, i.e. the CNN will learn the convolution filter weights that minimize the error. In the first layer of convolutions, primitive features like edges and colors are learned to be found. As convolution layers are stacked on top of each other, complex combinations of the features of previous layers are learned. With enough layers, things like faces and cars can be easily recognized.

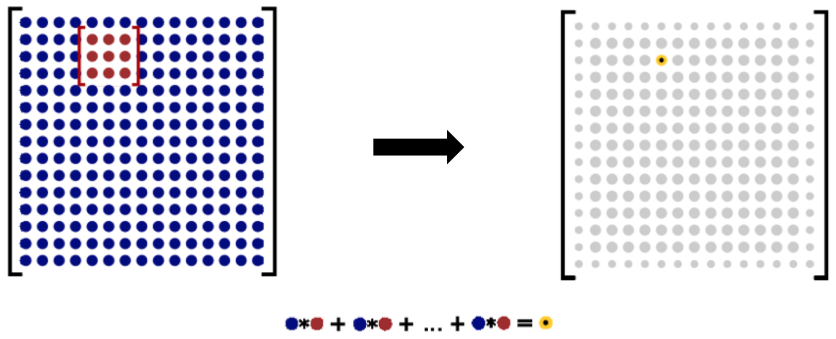


Figure 2.3. A convolution illustration. Credit: Standord CS231n (2015)

Normally after one or more convolution layers a pooling layer is used to decrease data size. The most common pooling operation is the 2×2, stride 2 kernel that performs the max operation. See Figure 2.4 for an illustration.

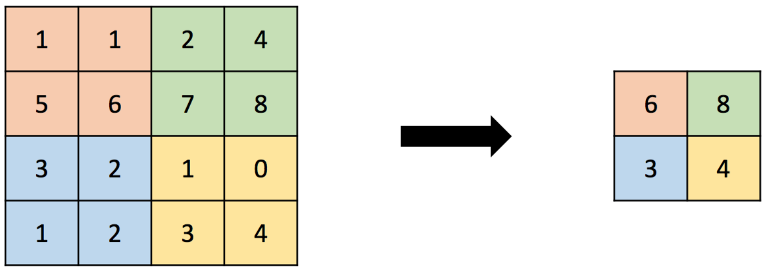


Figure 2.4. A pooling illustration

After successive convolution, activation, and pooling layers, one or more fully-connected layers are used, with one for output. This style of architecture for CNNs is very common. It can be easily generalized in the following form:

$$INPUT \rightarrow [[CONV \rightarrow ACT] * N \rightarrow POOL?] * M \rightarrow [FC \rightarrow ACT] * K \rightarrow FC$$

where N, M, K are positive integers, $INPUT$ indicates input data, $CONV$ indicates a convolution layer, ACT indicates an activation function, $POOL$ indicates a pooling layer, and FC indicates a fully-connected layer. The question mark after $POOL$ indicates that pooling is optional after each $CONV \rightarrow ACT$.

Figure 2.5 shows an example CNN with an image of an eye as input. There are two convolution layers with pooling after each ($N = 1, M = 2$) followed by one hidden fully-connected layer ($K = 1$) and one fully-connected output layer.

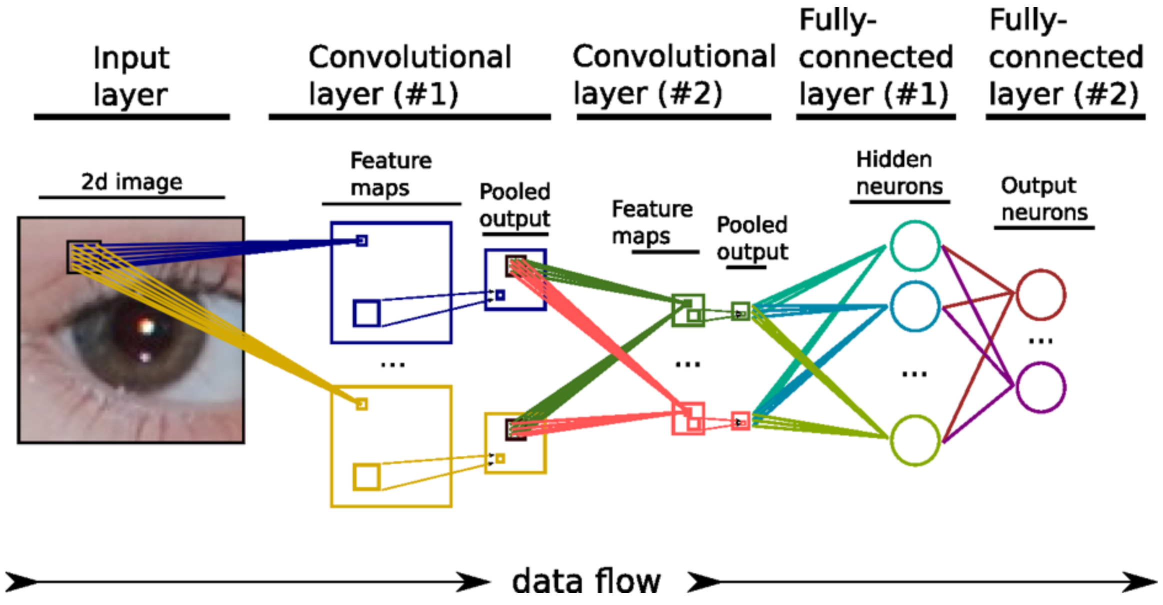


Figure 2.5. A convolutional neural network. Credit: Ryan Henning et al. (2014)

CNNs have been shown to provide state-of-the-art performance in many image recognition problems. A recent example is ResNet, which achieved a 3.57% classification error in ImageNet (He, Zhang, Ren, and Sun 2015a). This is impressive and is nearing human-level performance. It is because CNNs are so good at classifying image data that we use them for leukocoria detection.

CHAPTER THREE

Previous Work

Henning et al. developed neural networks to automate the detection of leukocoria in images (Henning, Rivas-Perea, Shaw, and Hamerly 2014). The authors achieved a classification accuracy of less than 3% on three classes: normal, leukocoric, and pseudo-leukocoric (a false leukocoria resulting from the white LED flash used by mobile phones). This chapter is a summary of the their work. See (Henning, Rivas-Perea, Shaw, and Hamerly 2014) for more detailed information.

3.1 Data

The data came from two sources: recreational photographs contributed by families of children with Rb and recreational photographs gathered from Flickr. The authors analyzed these images and extracted three types of eye crops: normal (437 eye images), leukocoric (222), and pseudo-leukocoric (173). The normal eyes came from both data sources, the leukocoric eyes came from the images of children with Rb, and the pseudo-leukocoric eyes came from Flickr. The authors performed ground-truth classification of each eye image. They also assumed that the images from Flickr do not contain true leukocoria, as leukocoria of the type we are seeking (a reflection off of the retina) is quite rare.

3.2 Experimentation and Results

The authors experimented on several neural network architectures, including traditional neural networks and convolution neural networks (CNNs). Network inputs were 40×40 raw RGB images. They used ten-fold cross validation on ten different crops of the data, with ten corresponding networks acting as an ensemble.

Networks 1 through 5 were traditional feed-forward fully-connected neural networks. They each used one hidden layer with a hyperbolic tangent squashing function. The number of hidden neurons was variable. Networks 6 through 20 were CNNs. They each used a kernel of size 5×5, a hyperbolic tangent squashing function, a 2×2 max-pooling layer after the first two convolution layers, and two fully-connected layers. The number of kernels per convolution layer, the number of convolution layers, and the number of fully-connected neurons were all variable. The authors used gradient descent with momentum (SGDM) to train networks 1-15 and RMSPROP for networks 16-20.

Table 3.1 contains the results of training. Network 16 performed the best with an error rate of $2.40 \pm 0.74\%$. This result is interesting because it has the least number of free parameters relative to the other networks. This indicates that a network with more capacity is not necessarily better than a network with less capacity for generalized performance.

id	Convolution Layers	Training Method	Fully-Connected Layers	Layer Types	# Free Parameters	Error Rate ± Std. Error
1	—	SGDM	6-3	h-s	28,827	6.37 ± 1.29%
2	—	SGDM	12-3	h-s	57,651	5.89 ± 1.19%
3	—	SGDM	25-3	h-s	120,103	6.49 ± 1.18%
4	—	SGDM	50-3	h-s	240,203	6.73 ± 1.13%
5	—	SGDM	100-3	h-s	480,403	6.97 ± 1.12%
6	7	SGDM	5-3	h-h-s	11,898	5.05 ± 0.88%
7	14	SGDM	5-3	h-h-s	23,767	5.05 ± 0.98%
8	21	SGDM	5-3	h-h-s	35,639	5.17 ± 0.97%
9	21	SGDM	10-3	h-h-s	69,679	5.41 ± 1.02%
10	21	SGDM	15-3	h-h-s	103,719	5.29 ± 0.91%
11	7-7	SGDM	5-3	h-h-h-s	3,502	3.97 ± 0.83%
12	14-14	SGDM	5-3	h-h-h-s	9,431	4.09 ± 0.76%
13	21-21	SGDM	5-3	h-h-h-s	17,810	3.73 ± 0.69%
14	21-21	SGDM	10-3	h-h-h-s	22,975	4.21 ± 0.66%
15	21-21	RMSPROP	15-3	h-h-h-s	28,140	4.33 ± 0.88%
16	7-7-7	RMSPROP	5-3	h-h-h-h-s	3,334	2.40 ± 0.74%
17	14-14-14	RMSPROP	5-3	h-h-h-h-s	11,545	2.88 ± 0.83%
18	21-21-21	RMSPROP	5-3	h-h-h-h-s	24,656	3.00 ± 0.83%
19	21-21-21	RMSPROP	10-3	h-h-h-h-s	25,621	2.88 ± 0.61%
20	21-21-21	RMSPROP	15-3	h-h-h-h-s	26,586	3.73 ± 0.64%

Table 3.1. Results from old neural networks

3.3 Mobile Deployment

The authors trained another neural network and deployed it in an iOS app and an Android app. These apps find faces and corresponding eyes in images, take crops of the eye, and feed the eye crop through the network. The user is notified when some eye appears to be leukocoric. There is a scanning mode that goes through existing photos and a screening mode that checks a screen-view video feed in real-time. Another requirement for our work is that these apps can be updated to use the newly trained neural networks.

3.4 Known Deficiencies

The results that Henning obtained are good, but there are also a number of known deficiencies that should be addressed.

Training Time A network that trains quickly allows the researcher to make rapid adjustments to the network with readily available results. If it takes too long to train, finding a good network becomes more time consuming. This system runs on the CPU and can take up to 36 hours to train the 10-network ensemble. As more and more training data is gathered, this training time would only get worse. Training time needs to be improved.

Difficulty of Use The training code is written in C++, is bare-bones, and not well documented. This makes it difficult to be able to make changes, add features, and rapidly prototype. It should be much easier to write network architecture and training code.

Performance While an error rate of 3% is good, it is reasonable to strive for better performance given the somewhat simple and yet very important subject matter. Consider that CNNs can achieve 99.8% accuracy on MNIST, which could possibly be considered to be similar in difficulty to leukocoria detection. Given more training data, we also want to make sure performance is well-generalized, with testing and validation accuracy very close to the training accuracy.

Mobile Network The CNN that was used on iOS and Android was documented but we are not sure what the performance actually is. The app network doesn't seem to work as well as one would expect given the error rates shown here. We need a network with documented performance that works better in the real world.

Classification Jitter The ten crops these networks trained on were not random. Intuitively, one would expect that these networks would favor eyes located nearby one of those ten crops. This behavior is indeed noticed in the CRADLE app deployed for iOS and Android. This behavior needs to be eliminated.

These deficiencies give us an open-ended, yet understandable problem to solve. We need to provide a system that trains new CNNs much more quickly, allows the user to make changes to network architecture and training code very easily, which will in turn allow CNNs with better performance to be trained.

CHAPTER FOUR

Implementation

In this chapter we discuss the work that we did on the improved system. We start with some early investigations which show the difficulty of using traditional approaches. We later show how these difficulties were overcome and discuss how our system improves over the existing one.

4.1 Early Investigations

We started by focusing on decreasing training time. Parallelizing network training was the clear way to do this. We collaborated with Dr. Grabow on this because of his expertise in parallel systems. We considered several different possible ways of doing this.

4.1.1 Naïve Parallelization

Given several CPU cores, we can give each core a subset of the training data to forward propagate in parallel, accumulate error, and backpropagate the error. The pseudocode looks like the following:

```
while not converged do
    for each minibatch in parallel do
        $err \leftarrow cnn.forward(minibatch)$
        $err\_total \leftarrow err\_total + err$
    end for
    $cnn.backprop(err\_total)$
    $err\_total \leftarrow 0$
end while
```

We investigated using OpenMP to do this parallelization for us automatically. Either way, there are a few problems with this approach. First, moderately extensive re-working of the existing code would be required. Second, we would still end up with a system that is difficult to change or update for future users. And finally, it's not obvious how much of a speed-up would be gained, or if it would even be worth the effort.

4.1.2 CUDA

We also considered writing our own CUDA kernels to do the convolutions, activations, pooling, etc. in parallel on a graphics processing unit (GPU). GPUs can have several thousand computation cores and are known to significantly speed up CNN training. The speed-up from training on a GPU would almost certainly be superior to our naïve parallelization. But, we would need to completely re-write all training and neural network code, and the resulting code would still be difficult to change or update for future users due to the low-level of C++ and CUDA kernels.

4.1.3 The Solution

During our investigation of how to speed up training, we discovered that there already existed several deep learning libraries which seem to fulfill our criteria for the new system. They took advantage of GPUs to processes training data in parallel, used scripting languages which makes making changes to the code relatively simple, and had dedicated open-source communities with promising support levels. Some of the libraries we investigated were Torch, Theano, Caffe, and deeplearning4j. We will discuss the one we picked in a future section.

4.2 Design

The system design is actually quite simple, as there are two distinct and primary things that need to be done by the system: data loading and repeated randomized CNN trials.

4.2.1 Data Loading

Vaclav Cibur developed a very nice image tagging and database platform called Facetag, from which we need to pull tagged examples of healthy and leukocoric eyes (Cibur 2016). The database uses PostgreSQL with the schema seen in Figure 4.1. The fields that are relevant for our work are described as follows:

image.id primary key for an image

image.data the image itself, encoded as a binary string

eye_tag.id primary key for an eye_tag

eye_tag.left, top, width, height together these define a bounding box around an eye

eye_tag.review_result indicates if an eye_tag has been reviewed or not

eye_tag.label indicates if an eye_tag is healthy or leukocoric

eye_tag.image_id foreign key to the image.id which corresponds to this eye_tag

Data loading needs to go through all eye_tags that have been accepted by a reviewer which are healthy or leukocoric, download the associated image, create a crop based on the bounding box, and save them to disk. We save to disk because generating these crops on-the-fly takes far too long, as moving around many gigabytes of images even within a single machine, can take a very long time.

Data augmentation artificially increases the amount of available training data. The idea behind it is to augment existing data in random ways so that the network becomes well-generalized i.e. doesn't prefer any particular image scale, rotation, etc. as input. This means we need create multiple variations of each training example, with random rotations, crops, scales, and flips.

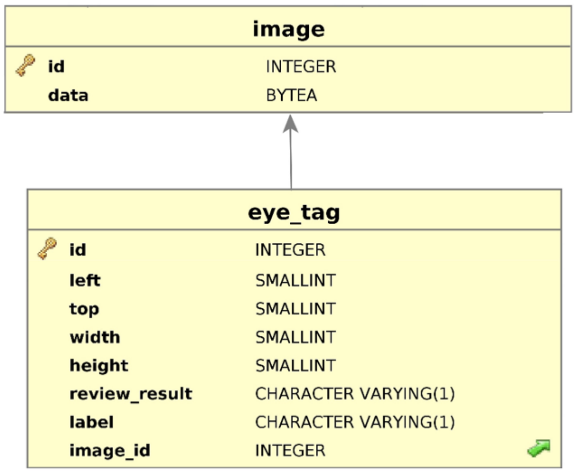


Figure 4.1. Relevant portion of the Facetag database schema

4.2.2 Randomized CNN Experiments

The end goal of the project is to easily find some CNN that maximizes generalized performance. Therefore, dozens if not hundreds of CNNs will need to be trained through a randomized experimentation process. That is, given some parameters to randomize and ranges for those random parameters, we should have a system that will generate and train a semi-random CNN and save the results for later viewing. Through this process we can see which models and parameters work well and which don't. Successive batches of experiments could then be run based on those models and parameters to produce even better CNNs. One could think of this as a guided evolutionary algorithm that may be repeated as desired.

Some network and learning parameters that should be randomized in this manner include:

- Maximum number of epochs
- Maximum number of epochs to try to find a new error minimum (for early stopping)
- Starting and minimum learning rate
- Learning rate decay rate, or saturation epoch

- Momentum
- Number of convolution layers
- Size of convolution kernels

Exactly what gets randomized will be discussed in the experimentation chapter.

4.3 Technologies Used

This section describes the technologies used in building the new system.

4.3.1 Docker

Docker is a program that automates the deployment of applications inside software containers. It provides a layer of abstraction and automation of operating-system-level virtualization. It uses the resource isolation features of the Linux kernel to allow independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines.

Docker *images* are state snapshots of an operating system. Docker *containers* are isolated operating systems spawned from some image which can be running or not running. There can be many containers spawned from a single image. Changes made to a container can be pushed to the parent image in a fashion similar to Git. Resources like GPUs and ports can be shared between containers simply by passing some parameters into a container start command. Docker is used for its ability to isolate concerns, share resources, and the ease with which it makes deployment.

4.3.2 Torch

Torch is a scientific computing framework with wide support for machine learning algorithms. It is easy to use and efficient, thanks to an easy and fast scripting language, Lua, and underlying C and CUDA implementations. It includes some important features like neural network and hyperoptimization libraries, ports to iOS and Android, and an open-source community. For these reasons, it is used in our system.

We tested the iOS and Android ports and the neural network and hyperoptimization libraries to verify they fulfill the requirements from Section 4.2 (with the addition of another database for storing hyperoptimizaiton results).

CNNs are very easy to build in Torch. Here's the Torch definition (using the Lua programming language) of Network 16 from Figure 3.1:

```
local net = nn.Sequential()

-- 3 input channels, 7 output channels, 5x5 kernel, 1x1 stride, tanh, 2x2
   pool, 2x2 stride
net:add(nn.SpatialConvolution(3, 7, 5, 5, 1, 1))
net:add(nn.Tanh())
net:add(nn.SpatialMaxPooling(2, 2, 2, 2))

-- 7 input channels, 7 output channels, 5x5 kernel, 1x1 stride, tanh, 2x2
   pool, 2x2 stride
net:add(nn.SpatialConvolution(7, 7, 5, 5, 1, 1))
net:add(nn.Tanh())
net:add(nn.SpatialMaxPooling(2, 2, 2, 2))

-- 7 input channels, 7 output channels, 5x5 kernel, 1x1 stride, tanh, 2x2
   pool, 2x2 stride
net:add(nn.SpatialConvolution(7, 7, 5, 5, 1, 1))
net:add(nn.Tanh())
net:add(nn.SpatialMaxPooling(2, 2, 2, 2))

-- re-size for fully-connected layers
net:add(nn.View(inputSize))

-- fully connected layer, 5 input channels, 3 output channels (classes)
net:add(nn.Linear(inputSize, 5))
net:add(nn.Linear(5, 3))
```

Training on the CPU is straightforward as well. Note that this is simply for illustration, as this is not how we trained the networks.

```
local criterion = nn.MSECriterion()
while not converged do
    for i = 1, #examples do
        -- feed it to the neural network and the criterion
        criterion:forward(net:forward(examples[i]), target[i])

        -- train over this example in 3 steps
        -- (1) zero the accumulation of the gradients
        net:zeroGradParameters()

        -- (2) accumulate gradients
        net:backward(input, criterion:backward(net.output, target[i]))

        -- (3) update parameters with a 0.01 learning rate
        net:updateParameters(0.01)
    end
end
```

4.3.3 PostgreSQL

The Facetag database stores the images on a PostgreSQL database, so we need to interface with that database to read images, eye tags, etc. The hyperoptimization library requires a PostgreSQL database to write hyperoptimization results. Therefore, we use two PostgreSQL databases in our system.

4.4 Deployment

This final section of the chapter is about how we deployed the system. Refer to Appendix A for the User Guide. See Figure 4.2 for a diagram of the whole system.

4.4.1 Docker and Hyperoptimization

The hyperoptimization container contains the hyperoptimization database, which stores the network training results.

4.4.1.1 Hyperoptimization Dockerfile. We choose to build using a Dockerfile, as it contains all the setup instructions necessary for Docker to create a Docker image. In the event that an image is lost, it can simply be recreated from a Dockerfile. Dockerfiles are only a handful of KB in size, which makes them nice to use with some version control system like Github.

4.4.1.2 Hyperoptimization Image. The hyperoptimzation image "hypero-db" is based on the vanilla Ubuntu 14.04 image. When "hypero-db" is built, it installs PostgreSQL, creates a database our hyperoptimization code is expecting, makes it accessible, and stops PostgreSQL.

4.4.1.3 Hyperoptimization Container. When container "hypero-db" is spooled up from image "hypero-db", all we do is tell it to restart PostgreSQL.

4.4.2 Docker and Learner

The learner container contains all of the Torch code necessary to fulfill the requirements from Section 4.2. It has links to the facetag database (which contains the images for training), the hyperoptimization database (which contains the network training results), and a host volume for simple data persistence. Many learners can be created, which can be run in parallel. Our training platform has two GPUs, so we create two learner containers.

4.4.2.1 Learner Dockerfile. We use a Dockerfile to build the learner for the same reasons we used it before.

4.4.2.2 Learner Image. The learner image "learner" is based on the vanilla CentOS 7 image. When "learner" is built, several things get installed: common dependencies like gcc, vim, git, etc., CUDA dependencies, CUDNN dependencies, Torch dependencies, CUDA itself, CUDNN itself, and Torch itself, useful Torch libraries, environment variables, and the PostgreSQL connection settings for "hypero-db".

4.4.2.3 Learner Container. When container "learner" is spooled up from image "learner", we setup a volume that is shared with the host, and link it to the "facetag-db" and "hypero-db" containers (which must already be running). We also attach the necessary Nvidia devices to make GPU access possible.

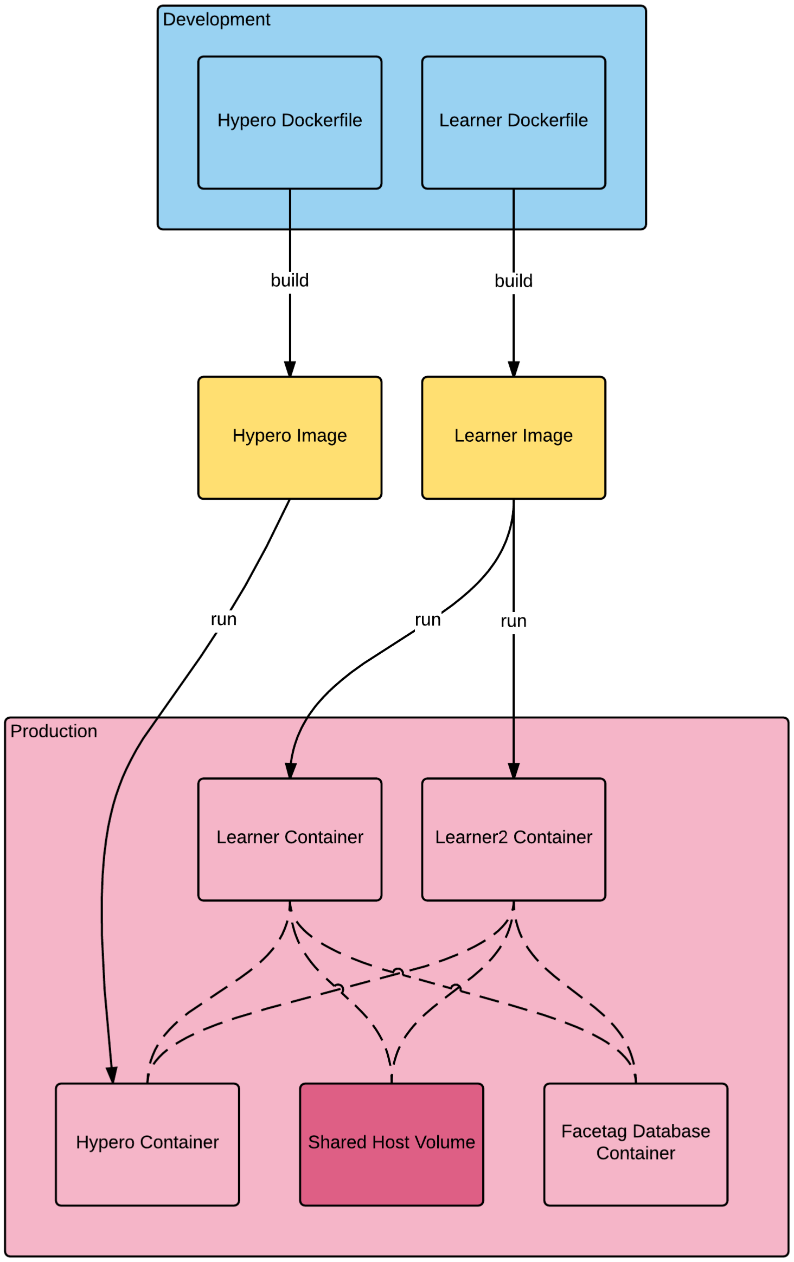


Figure 4.2. Deployment diagram; solid arrows indicate Docker commands, dashed lines indicate Docker links. The shared host volume is a link to a shared directory on the host, not a container, hence the different color.

CHAPTER FIVE

Improvements

The original leukocoria detector was trained in 2014. Since then, many advances have been made in deep learning research. Part of our project included reviewing the literature and attempt to improve network performance using these innovations and new rules-of-thumb. Unless otherwise stated, these concepts come from Stanford's CS231n Spring 2016 course, "Convolution Neural Networks for Visual Recognition" (Karpathy and Johnson 2016).

5.1 Convolutions

While there perhaps is no right or wrong answer when it comes to the use of convolutions, some concepts which are known to work well have been adopted as "best-practice."

5.1.1 Kernel Size and Stacks

The use of convolution kernels larger than 3×3 has fallen out of favor in CNNs. Consider the 3×3 convolution kernel with C channels. A 3×3 kernel will use $9C^2$ parameters with a receptive field of 3×3, a stack of two 3×3 kernels will use $18C^2$ parameters with a receptive field of 5×5, and a stack of three 3×3 kernels will use $27C^2$ parameters with a receptive field of 7×7. Compare this to a 7×7 filter with C channels, which would use $49C^2$ parameters with a receptive field of 7×7. Parameters can be saved by using stacks of 3×3 kernels rather than larger kernels. Stacks also provide the addition of more non-linearity. For these reasons, 3×3 kernels are by far the most common in practice.

5.1.2 *Padding*

Convolution kernels of size greater that 1×1 decrease the size of the data. For example, given a 40×40 image a 3×3 convolution operating on that image will result in a 38×38 image. This is not optimal if a deep network is desired. Therefore, padding is introduced, which is simply the addition of zeros around the border of the data. Given a convolution kernel stride of 1, the amount of padding needed is given by the formula $P = \lfloor (F - 1)/2 \rfloor$, where P is the amount of padding and F is the size of the convolution kernel. Padding is also likely to improve performance as information on the borders doesn't get washed away due to minimal coverage.

5.2 *Activation Functions*

Traditional CNNs used either the sigmoid ($\sigma(x) = 1/(1 + e^{-x})$) or the hyperbolic tangent ($tanh(x) = 2\sigma(2x) - 1$) as activation functions. However, there is a severe problem with both functions in CNNs: they saturate as the input x goes to $\pm\infty$. This means that the gradient at each end is near zero, resulting in the error gradient disappearing during error backpropagation. These neurons pass minimal gradient back to connected neurons, which in turn pass even less gradient back to their connected neurons. The deeper the network the bigger this problem becomes, as less and less gradient is passed back through the layers. Sigmoid also has the additional problem of the output not being zero-centered. Therefore, the hyperbolic tangent is always preferred over sigmoid in practice. For the rest of this section, we discuss new activation functions that outperform sigmoid and the hyperobolic tangent.

5.2.1 *ReLU*

The Recitified Linear Unit (ReLU) is defined as $f(x) = max(0, x)$. ReLUs have been shown to speed up training and improve network performance over sigmoid and the hyperbolic tangent. This is probably due to ReLUs being non-saturating and

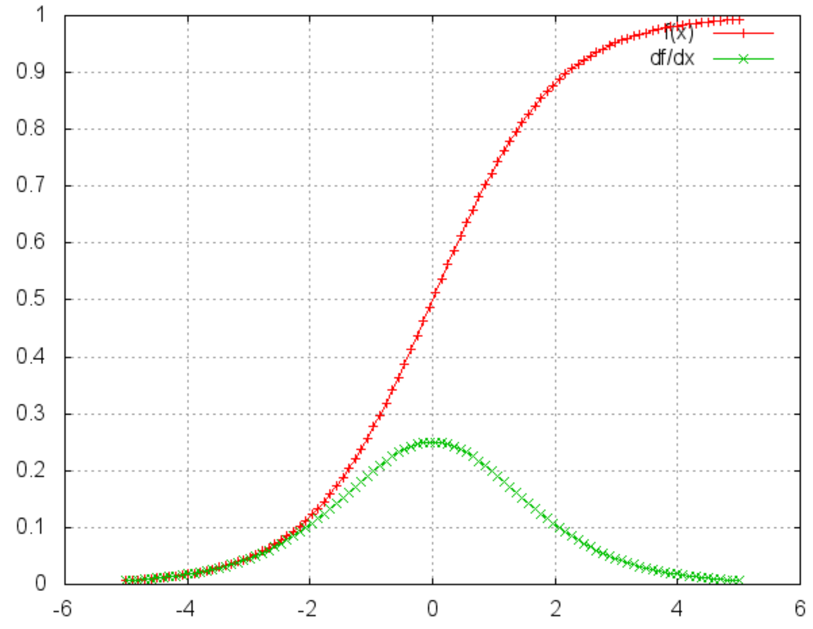


Figure 5.1. sigmoid

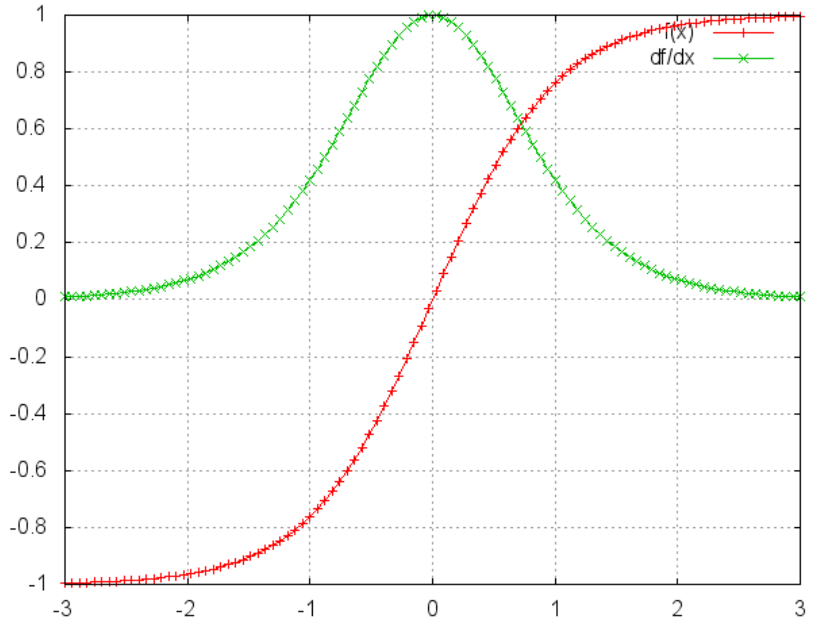


Figure 5.2. hyperbolic tangent

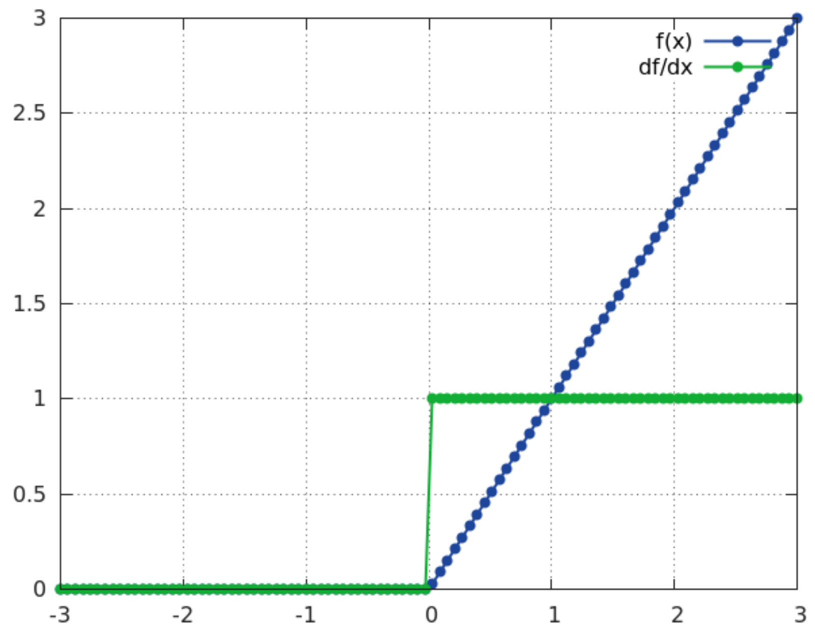


Figure 5.3. ReLU

constant gradient on unity on positive inputs, as well as being computationally trivial in both forward and backward directions (Krizhevsky, Sutskever, and Hinton 2012).

The biggest drawback of ReLUs is that on negative inputs there is no output on the forward pass and no gradient on the backward pass. This means that under certain conditions, some ReLUs in a network will never be activated and will never offer contributions to the weights of neurons that precede them. In fact, in some CNNs up to 40% of ReLU units are never activated, and therefore never useful for output on forward passes or updating weights on backward passes..

5.2.2 LReLU and PReLU

The Leaky Rectified Linear Unit (LReLU) is defined as $f(x) = max(0, x) + \alpha * min(0, x), \alpha \in [0, 1)$. LReLUs were introduced to solve the main problem with ReLUs (mentioned above) while retaining the advantages. The Parametric Rectified Linear Unit (PReLU) is the same as the LReLU, except that for PReLUs, α is a learnable parameter. LReLUs have been shown to outperform ReLUs (Maas, Hannun, and Ng

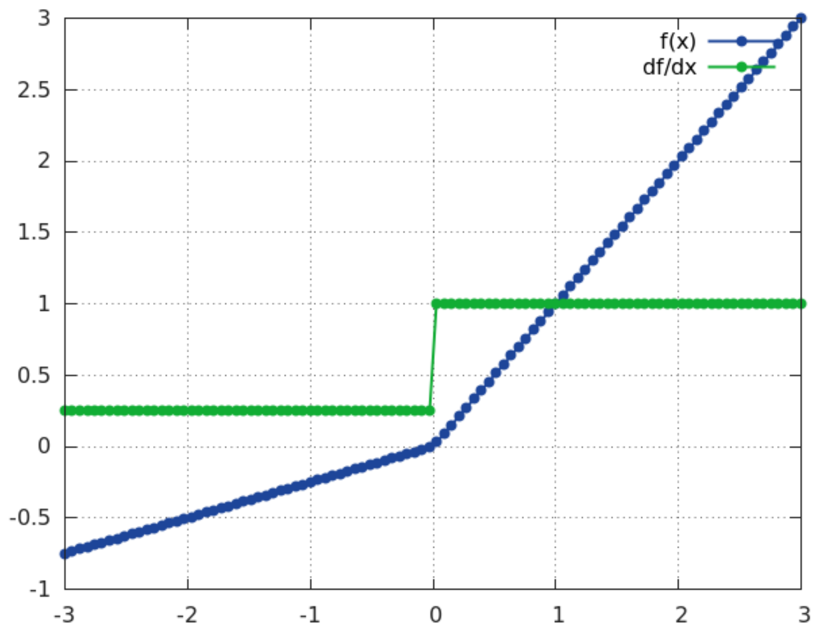


Figure 5.4. LReLU/PReLU, $\alpha = 0.5$

2013) and PReLUs have been shown to outperform both (He, Zhang, Ren, and Sun 2015b).

5.2.3 RReLU

The Randomized Rectified Linear Unit (RReLU) is defined as $f(x) = max(0, x) + \alpha * min(0, x), \alpha \sim U(l, u), l, u \in [0, 1)$. RReLUs have been shown to outperform all previously mentioned activation functions (Xu, Wang, Chen, and Li 2015).

5.2.4 ELU

The Exponential Linear Unit (ELU) is defined as $f(x) = max(0, x) + min(0, \alpha * (e^x - 1))$. One of the problems with LReLU, PReLU, and RReLU units concerns negative inputs: while passing some gradient back is good, saturation is actually a nice property. Details about why this is can be found in the original paper (Clevert, Unterthiner, and Hochreiter 2015), which also shows the ELUs can outperform all previously mentioned activation functions yet again.

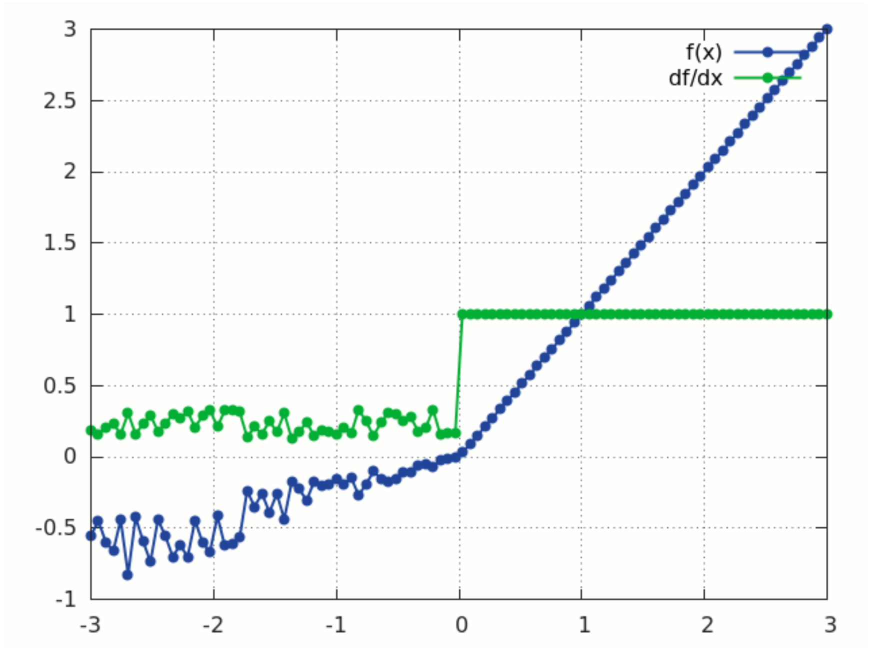


Figure 5.5. RReLU, $l = 0.125, u = 0.333$

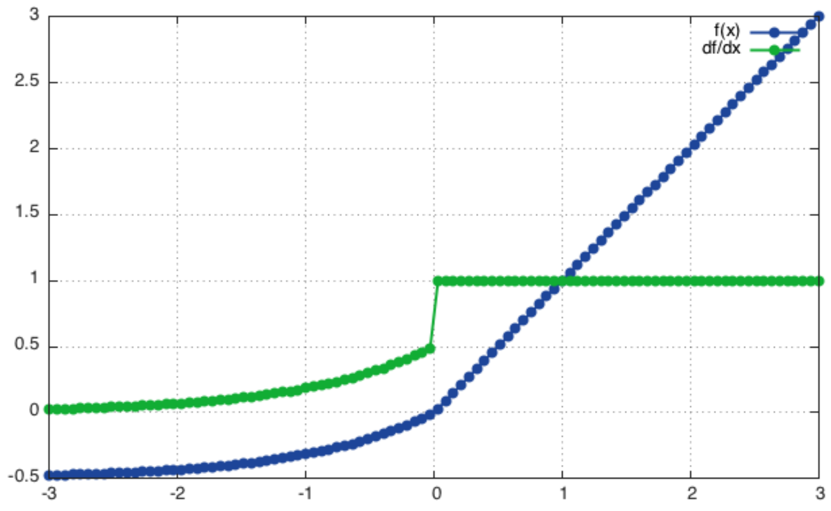


Figure 5.6. ELU, $\alpha = 0.5$

5.3 Pooling

Pooling is used to decrease the size of data as it moves through the network. With the introduction of convolution padding, pooling layers are the only layers that do this dimension reduction. The most common pooling operation is the 2×2 max pool with stride 2.

5.3.1 Overlapping Pooling

Sometimes a 3×3 max pool with stride 2 is used. In some cases the overlapping action that this pooling presents may be beneficial. Any larger max pooling is too destructive to be useful.

5.3.2 Convolution Pooling

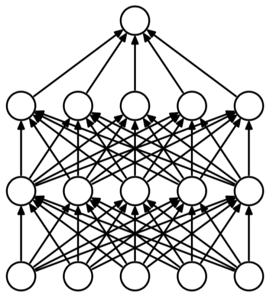
Some researchers don't like pooling because it destroys information. They avoid the use of pooling altogether simply by using convolution kernels of stride 2 in lieu of a traditional convolution-pooling pair. This is normally done with kernel size 2×2 and padding 0, or kernel size 3×3 and padding 1. State-of-the-art performance is achieved in some tasks with this configuration (Springenberg, Dosovitskiy, Brox, and Riedmiller 2014).

5.4 Other

5.4.1 Dropout

One serious problem with CNNs is their tendency to over-fit and develop neuron co-adaptation during training. Dropout is a technique designed to mitigate this.

The idea is to randomly not use some neurons and their connections during training. At training time, each neuron in layer i has probability p_i of not being used. During evaluation, the output of layer i is multiplied by p_i to prevent saturation. Intuitively, training is done on a randomly sampled subset of the original network, resulting in a sort of built-in ensemble. See Figure 5.7 for a graphical representation of dropout.


(a) Before dropout

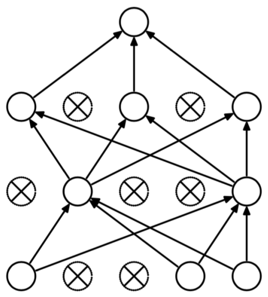

(b) After dropout

Figure 5.7. Graphical representation of dropout. Credit: blog.christianperone.com

Dropout has been shown to be useful in most domains, including image classification. For example, dropout has been used to improve upon already state-of-the-art performance on SVHN, ImageNet, CIFAR, and MNIST (Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov 2014).

5.4.2 Weight Initialization

Proper weight initialization can mean the difference between good performance and state-of-the-art performance. Kaiming initialization takes advantage of properties of rectifier-based networks in order to boost performance over more traditional initializations (He, Zhang, Ren, and Sun 2015b).

CHAPTER SIX

Experimentation

6.1 Method

With the new system implemented, we move on to experimental results. We use the same data as discussed in Chapter 3.

6.1.1 Round 1

For the first round of experiments, we desire to provide a baseline for performance, so we make several changes to provide that reasonable baseline: the two classes are set to be evenly represented, the various crops from the previous work are not used, nor is the corresponding CNN ensemble. As detection of pseudo-leukocoria is no longer important, we ignore the images corresponding to that class. All of these slight changes result in 297 examples of leukocoria and 297 examples of normal eyes. The validation and testing sets each get 10% of the data.

We set up a battery of 600 CNNs to train, validate, and test. Because we have two graphics cards, each is responsible for 300 CNNs. Each CNN is trained using mini-batch gradient descent with momentum for a minimum of 50 epochs and a maximum of 250. Convolutions of size 3×3 were used, as was padding. The entire battery took about 3 hours, compared to the several hours it took to train a single network using the same data on the old system.

Each of the 600 CNNs is built from many randomly selected values. This allows us to search for the CNN that performs the best for the given problem. The random values are selected from ranges as follows:

Start Learning Rate The learning rate for the first epoch, selected from $U(0.001, 1)$.

Minimum Learning Rate The minimum learning rate as a fraction of the start learning rate, selected from $U(0.001, 1)$.

Momentum The momentum coefficient, selected from $U(0.0, 0.9)$.

Saturation Epoch The epoch at which the learning rate becomes the minimum learning rate (linear decay), selected from $N(250, 50)$.

Max Out Norm Normalizes the neuron weights as seen in (Hinton, Srivastava, Krizhevsky, Sutskever, and Salakhutdinov 2012), selected from $\{x|1 \leq x \leq 4, x \in \mathbb{N}\}$.

Convolution Stacks The number of convolutions before pooling (N from Chapter 2), selected from $\{x|1 \leq x \leq 2, x \in \mathbb{N}\}$.

Start Convolution Filters The number of convolution filters on the first layer was fixed to 8.

Final Convolution Filters The number of convolution filters on the last layer, with intermediate layers increasing linearly from start to final, selected from $\{x|3 \leq x \leq 32, x \in \mathbb{N}\}$.

Convolution Layers The number of convolution layers (M from Chapter 2), selected from $\{x|2 \leq x \leq 4, x \in \mathbb{N}\}$.

Activation The activation function to use, selected uniformly from {Relu, PReLU, RReLU, ELU}.

Pool Size The size of the pooling filter, selected from $\{x|2 \leq x \leq 3, x \in \mathbb{N}\}$.

Pool Method The pooling method to use, selected uniformly from {Conv, Max}.

FC Layers The number of fully-connected layers to use (K from Chapter 2), selected from $\{x|1 \leq x \leq 2, x \in \mathbb{N}\}$.

FC Neurons The number of neurons in each fully-connected layer, selected from $\{x|10 \leq x \leq 100, x \in \mathbb{N}\}$.

FC Dropout If dropout is used (which itself is given a probability of 50%), the value of p for dropout on the fully connected layers, selected from $U(0.2, 0.5)$.

6.1.2 Round 2

For the second round of experiments, we determine how valuable more data is. Therefore, we include all of the crops from the previous work, which means a 10x increase in training data. We set up another battery of 600 CNNs to train, validate, and test, all using the same criteria and randomly assigned values as before except for the following additions:

Convolution Kernel Size The size of the kernels, selected uniformly from {3, 5}.

Convolution Dropout If dropout is used (which itself is given a probability of 50%), the value of p for dropout on convolution layers, selected from $U(0.1, 0.2)$.

6.2 Results

The 50 top-performing networks for both batteries are contained in Figure B.1 and Figure B.2. We see that even with minimal data we can train a single network that is nearly as good as the entire ensemble from the previous work (96.6% vs 97.6% accuracy). As ensembles typically add 2-3% to classification accuracy (Karpathy and Johnson 2016), this shows that the improvements we added are indeed improving network performance. Using the extra crops sees a network with an accuracy of 99.5%, which shows that we are indeed data limited.

Despite these good results, we are still skeptical of good real-world performance. An example set of 297 unique eyes per class is not much, even with many variations, and likely cannot be well-generalized to the real-world. We are curious to see how much performance increases as more data becomes available and used.

CHAPTER SEVEN

Conclusion

We have developed a new system which makes training CNNs much easier than before. The system uses GPUs to train CNNs significantly faster than before. The use of a scripting language (Lua) and machine learning framework (Torch) allows changes to be made quickly and easily. We leverage and expand existing Torch libraries to make training hundereds of semi-random CNNs quite simple. Our deployment on Docker allows simple, compartmentalized, and platform-agnostic development deployment.

The networks we trained on this system used recent innovations in deep learning research. The results are promising, with a single network not using data augmentation achieving 96.6% accuracy and another single network using data augmentation achieving 99.5% accuracy.

CHAPTER EIGHT

Future Work

8.1 More data

Machine learning performance is heavily dependent on having lots of good data. More batteries of CNNs need to be trained when more data becomes available. We expect an accuracy of 99.8% to be feasible (because this problem is likely not harder than MNIST, whose state-of-the-art accuracy is 99.8%).

8.2 More CNN innovations

There is an almost endless amount of new things to try to improve CNN performance. Here is a small list of examples that would be interesting to investigate:

- Layer-sequential unit-variance initialization, which nearly achieves SOTA in CIFAR-10 (Mishkin and Matas 2015)
- Fractional max pooling, which achieves SOTA in CIFAR-10 (Graham 2014)
- Generalized max-average pooling, which achieves SOTA in SVHN (Lee, Gallagher, and Tu 2015)
- Dilated convolutions are mainly useful for dense captioning, but may be useful here (Yu and Koltun 2015)
- Local contrast normalization, a classic and possibly helpful pre-processing technique
- Interesting network architectures like ResNet, Network-In-Network, or Inception

8.3 Surprisingly, Torch is no longer king (or viable)

With the release of TensorFlow in late 2015, Torch seems to be losing favor in the deep learning community. When we started working with Torch, Android and iOS ports were both working. Unfortunately, support for those ports has effectively been dropped, as neither one has been in good working condition since December 2015. This is despite reasonable indications that they would be supported. TensorFlow has a working Android port and an upcoming iOS port. It has a significantly larger contributor base despite it's much younger age (see `https://github.com/tensorflow/tensorflow` vs `https://github.com/torch/torch7`), and has the full support of Google. For these reasons, much of the work in this project must be replicated (and preferably made even better) using TensorFlow.

APPENDICES

APPENDIX A

User Guide

Spending hours getting someone else's code to run can be quite frustrating. Thankfully, a lot of work was put into making it as easy as possible for future developers to get working on this project. This guide explains how to set up and use the new system, which should be relatively painless.

A.1 Source Code

Start by getting the source code.

(1) As this project is designed to run on the Facetag server, `ssh` into it.

(2) Change the working directory to wherever the user desires the source code to be. The author uses `/home/grad/boer/shared` for reasons that will be evident later.

(3) Run `git clone https://github.com/boerjames/leuko`.

A.2 Hyperoptimization Database

Now that the code is obtained, set up the hyperoptimization database by doing the following.

A.2.1 Build the image

(1) If necessary, change the working directory to `leuko/docker/hypero`.

(2) Run `./docker-build.sh`. This step may take multiple tries to complete successfully, as Docker still has some rough edges.

(3) Run `docker images` to verify the existence of the "hypero-db" image.

A.2.2 Run the container

(1) If necessary, change the working directory to `leuko/docker/hypero`.

(2) Run `./docker-run.sh`.

(3) Run `docker ps -a` to verify the existence of the "hypero-db" container. If it isn't running, run `docker start hypero-db`.

A.3 Learner

Now that the hyperoptimization database is setup, continue by setting up the learner.

A.3.1 Build the image

(1) If necessary, change the working directory to `leuko/docker/learner`.

(2) Run `./docker-build.sh`. This step may take multiple tries to complete successfully, as Docker still has some rough edges.

(3) Run `docker images` to verify the existence of the "learner" image.

A.3.2 Run the container

(1) If necessary, change the working directory to `leuko/docker/learner`.

(2) Run `./docker-run.sh SHARED NAME`. The argument `SHARED` is the directory from the host OS that will be readable from the container. This is useful for persistent data. The argument `NAME` is the name of the container that will be run. The author typically runs the following: `./docker-run.sh /home/grad/boer/shared learner`. If multiple learner containers are desired, most likely for training using both GPUs, create another container with the same shared directory but a different name.

(3) Run `docker ps -a` to verify the existence of the container. If it isn't running, run `docker start NAME`.

A.4 Clean Up

Sometimes Docker will leave behind broken or dangling image layers. To clean these up do the following:

(1) Change the working directory to `leuko/docker`.

(2) Run `./docker-clean.sh`.

(3) Be careful with this, as all unused layers will be permanently deleted from the cache.

A.5 Train CNNs

Now that all the necessary components are built, we can train some CNNs.

A.5.1 Download Crops

Start by downloading crops from the Facetag database.

(1) If necessary, attach a "learner" container by running `docker attach NAME`, where `NAME` is the name of the container.

(2) If necessary, change the working directory to `leuko`.

(3) Run `th DownloadCrops.lua --password PASSWORD`, where `PASSWORD` is the password to the Facetag database. Ask Dr. Hamerly for this, as it is nowhere to be found in our code on purpose.

    (a) The script will run for at least an hour as there are several hundred gigabytes of data to download in order to make the small crops.

    (b) Inspect `DownloadCrops.lua` to see which command line options are available.

    (c) When the script is complete, the crops will be available at `/root/shared/data/normal` and `/root/shared/data/leuko`.

A.5.2 Build Data Source

Now that the crops are available, we can build a usable data set.

(1) If necessary, attach a "learner" container by running `docker attach NAME`, where `NAME` is the name of the container.

(2) If necessary, change the working directory to `leuko`.

(3) Enter the Torch REPL by running `th`.

(4) Run `require './DataLoader.lua'`.

(5) Run `ds = DataLoader.loadData(DATA_PATH, DATA_SIZE, EQUAL_REPRESENTATION, TEST_PERCENTAGE, VALID_PERCENTAGE, VERBOSE)`, where the arguments should be set as necessary. The author typically uses `DATA_PATH = '/root/shared/data'`, `DATA_SIZE = {3,40,40}`, `EQUAL_REPRESENTATION = true`, `TEST_PERCENTAGE = 0.15`, `VALID_PERCENTAGE = 0.15`, and `VERBOSE = true`.

(6) Run `torch.save('datasource.t7', ds)` to save the data set.

(7) Exit the Torch REPL by running `os.exit()`.

(8) Move `datasource.t7` by running `mv datasource.t7 /root/shared/data`.

A.5.3 Run Experiments

Now that we have prepared a data source, we can run lots of experiments. This section can be done twice using a second "learner" container to use both GPUs.

(1) If necessary, attach a "learner" container by running `docker attach NAME`, where `NAME` is the name of the container.

(2) If necessary, change the working directory to `leuko`.

(3) Run `Hypero.lua --useDevice DEVICE`, where DEVICE is the GPU to use, either 1 or 2 as Lua is 1-indexed.

  (a) There are many command line parameters to change, please see the file to see what can be changed. Typically the author changes the command line options in the source code itself.

  (b) This script will train however many CNNs are specified, which can take seconds, minutes, hours, or days. It may be possible to set up a script that notifies the user via email when the training is complete, but we had difficulty getting this to work due to restrictions on the host machine.

  (c) Results such as the saved networks can be found in `/root/shared/log`, `/root/shared/unit`, `/root/shared/save`. Delete these directories as necessary to save space on the server.

A.5.4 Retrieve Results

When the experiments are complete, retrieve the results from the hyperoptimization database.

(1) If necessary, attach a "learner" container by running `docker attach NAME`, where `NAME` is the name of the container.

(2) If necessary, change the working directory to `hypero`.

(3) Run `th scripts/export.lua --batteryName BATTERY_NAME --versionDesc VERSION_DESC --metaNames 'hostname' --resultNames 'trainAcc,validAcc,testAcc' --orderBy 'testAcc' --desc`, where `BATTERY_NAME` is the name of the battery and `VERSION_DESC` is the name of the version from when `Hypero.lua` was run.

(4) A file `hyper.csv` with the sorted results of the experiments is created. Use a program like Cyberduck to easily view it from a remote computer.

A.6 Complete

This concludes the section on how to get started using the new learning platform for the leukocoria detection project. Enjoy!

APPENDIX B

Results

As the table of experimentation results is too large to fit in the main text, we show it here.

hexId	satEpoch	startLR	minLR	momentum	maxOutNorm	activation	startConvolutionFilters	finalConvolutionFilters	convolutionStacks	numConvolutionLayers	poolMethod	poolSize	dropout	fcDropoutProb	numFCLayers	numFCNeurons	trainAcc	validAcc	testAcc
1330	311.0183382	0.08664861	0.000367362	0.235514973	2	PReLU	8	25	1	2	Max	3	TRUE	0.449836982	2	44	0.81512605	0.93220339	0.966101695
1356	187.5208854	0.202705933	0.092268488	0.427572363	4	ReLU	8	9	2	4	Conv	2	FALSE	0.272208137	2	44	0.905462185	0.949152542	0.966101695
1240	213.6346912	0.283124349	0.002534267	0.523135812	1	ReLU	8	24	1	2	Max	3	FALSE	0.340276113	1	100	0.754201681	0.915254237	0.949152542
1274	254.674558	0.008382632	8.62E-06	0.718215575	4	ELU	8	13	2	2	Max	2	FALSE	0.286491155	2	36	0.897058824	0.966101695	0.949152542
1019	275.3695583	0.003996147	0.003207279	0.626159006	4	RReLU	8	20	2	3	Max	3	FALSE	0.432442524	1	58	0.771008403	0.93220339	0.949152542
1042	272.0075339	0.114437759	0.064461206	0.178636424	2	RReLU	8	9	2	4	Conv	2	TRUE	0.200131979	2	16	0.888655462	0.966101695	0.949152542
1040	260.9178224	0.01824492	0.000300447	0.608124132	2	PReLU	8	19	1	3	Conv	3	FALSE	0.351659019	1	70	0.911764706	0.949152542	0.949152542
1273	264.5962948	0.142004225	0.007994054	0.53088616	2	RReLU	8	19	1	3	Conv	3	TRUE	0.248036839	1	36	0.932773109	0.966101695	0.949152542
941	262.9314895	0.072897218	0.000335597	0.63795725	3	ReLU	8	13	2	3	Max	3	TRUE	0.215758438	2	45	0.926470588	0.949152542	0.949152542
848	251.2699072	0.114606613	0.027776248	0.678048298	2	RReLU	8	13	2	2	Conv	3	TRUE	0.349780902	1	67	0.798319328	0.949152542	0.949152542
899	157.7161863	0.068481152	0.025112897	0.856476526	3	ReLU	8	27	2	4	Max	3	FALSE	0.344112852	1	27	0.899159664	0.966101695	0.949152542
985	301.2438543	0.057804935	0.010470742	0.216184287	4	RReLU	8	30	2	3	Max	3	FALSE	0.342066316	2	94	0.962184874	0.966101695	0.949152542
914	242.4247986	0.115938375	0.010216256	0.09718976	4	PReLU	8	16	1	3	Max	2	TRUE	0.394803516	1	87	0.869747899	0.949152542	0.949152542
1304	200.3966019	0.024935535	9.55E-05	0.635460232	2	ELU	8	24	1	2	Conv	2	FALSE	0.302809152	1	22	0.93697479	0.949152542	0.949152542
1078	269.1369243	0.044642948	8.26E-05	0.599374606	2	PReLU	8	24	2	3	Max	3	FALSE	0.408433686	2	37	0.903361345	0.966101695	0.949152542
1081	231.7609995	0.082169759	0.003455411	0.461680522	3	RReLU	8	28	2	3	Max	2	TRUE	0.377686816	1	54	0.945378151	0.966101695	0.949152542
859	264.8091225	0.013662654	0.003056698	0.454998501	2	PReLU	8	29	2	3	Max	3	FALSE	0.201401811	2	83	0.859243697	0.93220339	0.93220339
992	249.1952009	0.046827443	0.000277386	0.228045749	2	ReLU	8	25	2	3	Conv	3	TRUE	0.479695984	1	12	0.911764706	0.949152542	0.93220339
1346	145.2126977	0.046418221	0.001411609	0.032034246	2	ReLU	8	20	2	3	Conv	3	FALSE	0.432005943	2	45	0.785714286	0.949152542	0.93220339
1138	163.7863822	0.233123918	0.007937216	0.088424425	3	PReLU	8	11	1	4	Max	3	TRUE	0.305253306	1	95	0.915966387	0.93220339	0.93220339
939	233.6121241	0.138050376	0.002101428	0.199332087	2	ReLU	8	24	1	3	Max	2	FALSE	0.242466799	2	96	0.871848739	0.915254237	0.93220339
1101	262.7993386	0.152719071	0.000532623	0.896752691	2	PReLU	8	15	1	4	Conv	3	TRUE	0.329518418	2	47	0.855042017	0.966101695	0.93220339
829	223.4579257	0.037362911	0.000774847	0.873060793	3	RReLU	8	25	1	3	Conv	2	FALSE	0.341080004	1	68	0.901260504	0.949152542	0.93220339
894	260.8170135	0.108195779	0.000791958	0.328524114	4	PReLU	8	13	1	2	Max	3	FALSE	0.225077666	2	89	0.918067227	0.966101695	0.93220339
887	296.7628352	0.044213039	0.010911822	0.743520128	2	RReLU	8	29	2	4	Conv	3	FALSE	0.430848194	1	94	0.863445378	0.966101695	0.93220339
1246	220.4709943	0.037503997	0.02855196	0.824040348	3	ELU	8	13	1	4	Conv	3	TRUE	0.320128252	2	96	0.941176471	0.949152542	0.93220339
828	280.1996544	0.04199416	0.004381524	0.243249133	2	PReLU	8	18	2	4	Max	3	TRUE	0.352330085	2	76	0.827731092	0.966101695	0.93220339
1351	328.1443304	0.049103555	9.79E-05	0.609236204	4	ELU	8	19	1	2	Conv	2	TRUE	0.362543731	2	14	0.913865546	0.949152542	0.93220339
925	170.2675189	0.061690275	0.000153933	0.333747884	2	RReLU	8	21	1	3	Conv	2	FALSE	0.369717053	2	95	0.892857143	0.966101695	0.93220339
1350	248.696652	0.036059319	0.000116218	0.469373106	4	PReLU	8	29	2	2	Max	2	TRUE	0.475193735	1	51	0.871848739	0.966101695	0.93220339
935	237.8381417	0.18767672	0.096697453	0.251050854	4	ReLU	8	8	2	2	Max	2	TRUE	0.330647922	2	23	0.880252101	0.966101695	0.93220339
1271	248.6256382	0.306203419	0.002607777	0.778554839	1	PReLU	8	23	1	4	Max	3	FALSE	0.462915133	1	44	0.781512605	0.93220339	0.93220339
1004	252.5953789	0.189094194	0.024445033	0.104993306	3	PReLU	8	12	1	3	Conv	2	TRUE	0.34267154	1	91	0.882352941	0.966101695	0.93220339
989	373.8001295	0.027694355	0.000652937	0.083503303	3	ELU	8	19	2	2	Max	2	TRUE	0.316059727	2	94	0.888655462	0.949152542	0.93220339
923	217.3316545	0.007078696	3.74E-05	0.300877697	4	ELU	8	29	1	4	Max	2	TRUE	0.423289557	1	21	0.827731092	0.93220339	0.93220339
1186	255.1703252	0.010572303	0.000640313	0.280480286	2	RReLU	8	24	2	3	Conv	3	FALSE	0.247179678	1	20	0.783613445	0.949152542	0.93220339
1397	281.699976	0.028565762	0.009481878	0.739434986	2	ELU	8	20	2	2	Conv	2	FALSE	0.380677074	1	85	0.932773109	0.93220339	0.93220339
895	190.8012299	0.001396368	1.47E-05	0.088292122	4	ELU	8	27	2	3	Conv	2	FALSE	0.492838846	1	90	0.848739496	0.949152542	0.93220339
1272	158.6333941	0.031532844	0.012380631	0.859216882	2	RReLU	8	14	1	2	Conv	2	TRUE	0.212403927	1	60	0.897058824	0.93220339	0.93220339
1227	273.1455967	0.048201299	0.002509373	0.721744428	3	ELU	8	8	2	4	Conv	2	FALSE	0.334243921	2	61	0.901260504	0.966101695	0.93220339
1208	158.6861579	0.124258581	0.002504448	0.695067397	3	RReLU	8	29	2	4	Max	3	TRUE	0.313943298	1	81	0.930672269	0.949152542	0.93220339
1198	285.0150992	0.121555928	0.002072066	0.259213235	4	PReLU	8	32	1	3	Max	2	TRUE	0.489024582	2	26	0.911764706	0.966101695	0.93220339
1179	295.8860317	0.055110315	0.004056306	0.588717655	1	ELU	8	32	1	2	Conv	3	FALSE	0.285874585	1	72	0.951680672	0.949152542	0.93220339
1374	227.614117	0.004508949	2.81E-05	0.647590425	3	PReLU	8	27	2	2	Conv	3	FALSE	0.452934997	1	70	0.756302521	0.898305085	0.915254237
870	297.6883191	0.069754319	0.00078843	0.611545392	2	ReLU	8	16	2	2	Conv	3	FALSE	0.397891986	1	59	0.773109244	0.93220339	0.915254237
918	265.6591252	0.087756401	0.01605421	0.627533335	2	ReLU	8	13	1	2	Max	3	TRUE	0.420194874	1	83	0.911764706	0.966101695	0.915254237
1183	200.2725473	0.005886628	0.000524279	0.057155636	3	PReLU	8	31	1	3	Conv	3	FALSE	0.407245833	2	22	0.640756303	0.898305085	0.915254237
880	303.8255841	0.022140123	0.011753715	0.183705231	3	PReLU	8	21	1	3	Max	3	FALSE	0.305228893	1	40	0.718487395	0.966101695	0.915254237
1359	270.988096	0.053893898	0.000194458	0.71038198	2	PReLU	8	29	2	2	Conv	2	TRUE	0.480401802	2	29	0.892857143	0.93220339	0.915254237
928	241.5161719	0.045426553	0.016302366	0.408462217	3	RReLU	8	16	2	4	Conv	2	FALSE	0.365802632	1	76	0.913865546	0.966101695	0.915254237

Table B.1. Results without extra crops

hexId	satEpoch	startLR	minLR	momentum	maxOutNorm	activation	convolutionKernelSize	convDropoutProb	startConvolutionFilters	finalConvolutionFilters	convolutionStacks	numConvolutionLayers	poolMethod	poolSize	dropout	fcDropoutProb	numFCLayers	numFCNeurons	trainAcc	validAcc	testAcc
572	235.2479838	0.378651091	0.235757551	0.642607996	4	RReLU	3	0.127220721	9	32	2	4	Max	2	TRUE	0.484395624	1	70	0.976827094	0.995006242	0.995422389
55	175.9589063	0.55092653	0.498024084	0.486126684	1	PReLU	5	0.167908004	10	28	2	4	Max	3	TRUE	0.370078446	2	24	0.974420677	0.993757803	0.992093217
451	205.9315649	0.607343804	0.003554934	0.892360449	3	PReLU	5	0.168426745	8	31	2	4	Ave	3	TRUE	0.336436674	1	50	0.979857398	0.992509363	0.993757803
344	192.5969792	0.158396045	0.024511531	0.242248226	3	RReLU	5	0.116539283	8	31	2	4	Conv	3	FALSE	0.347241228	2	86	0.981105169	0.990428631	0.989180191
189	222.5527992	0.513529207	0.293113798	0.689570211	4	PReLU	5	0.174325585	12	12	1	4	Conv	3	FALSE	0.309909921	2	108	0.980659537	0.990012484	0.987515605
81	186.4415795	0.189135657	0.000220765	0.64318643	2	RReLU	3	0.111140832	10	32	2	4	Max	3	TRUE	0.303158657	2	80	0.9614082	0.989180191	0.986267166
355	135.7480121	0.574477336	0.001187626	0.417044525	2	RReLU	3	0.12974231	11	11	1	4	Max	3	FALSE	0.393311178	1	92	0.956327986	0.988347898	0.98460258
161	136.1371818	0.542762106	0.014599195	0.235067373	1	ReLU	5	0.198466846	8	9	2	4	Max	2	FALSE	0.325743997	1	60	0.956773619	0.988347898	0.986267166
151	113.2564257	0.196990105	0.005627606	0.189321877	4	PReLU	5	0.160438913	9	9	2	3	Max	3	TRUE	0.457321036	1	88	0.951247772	0.987931752	0.981689555
560	170.1376626	0.168605744	0.000353951	0.532105332	2	RReLU	5	0.149514891	11	28	2	2	Max	3	TRUE	0.4414414	1	10	0.965864528	0.987099459	0.983354141
429	159.0813088	0.140465366	0.017461991	0.267001746	3	PReLU	5	0.103143469	10	10	2	2	Max	2	TRUE	0.435851031	2	49	0.965953654	0.986683313	0.981689555
483	133.0434481	0.546060803	0.045336788	0.674684824	4	PReLU	3	0.105879765	8	10	2	3	Ave	3	TRUE	0.412097337	1	88	0.959625668	0.986267166	0.984186434
591	146.5839579	0.452431063	0.006151972	0.486115516	3	PReLU	3	0.148294122	10	19	1	4	Max	3	FALSE	0.348319071	2	120	0.951960784	0.986267166	0.979608822
551	149.67593	0.265522112	0.093458207	0.50369725	1	PReLU	3	0.186084064	8	25	1	3	Conv	3	FALSE	0.473901271	1	124	0.976024955	0.985434873	0.983354141
363	202.2580142	0.101055087	0.027356789	0.587147482	3	ReLU	5	0.116726517	9	28	2	3	Max	2	FALSE	0.305475943	1	15	0.960249554	0.985018727	0.983354141
615	147.1214064	0.182402531	0.000495867	0.431694925	2	ReLU	3	0.115956375	9	23	1	4	Conv	3	TRUE	0.46968652	2	16	0.954010695	0.982521848	0.978360383
213	283.0705217	0.053962391	0.015398742	0.592259298	4	ReLU	5	0.124374492	9	20	2	3	Conv	3	FALSE	0.339218962	1	21	0.958645276	0.982105701	0.97752809
102	204.3887433	0.900595991	0.002668172	0.825896987	2	RReLU	3	0.187242597	9	32	1	3	Conv	3	FALSE	0.363511024	2	36	0.970677362	0.981273408	0.982937994
49	147.703966	0.113487388	0.000470506	0.073801326	3	ReLU	5	0.184599566	9	28	2	4	Max	3	FALSE	0.498172623	2	101	0.942691622	0.981273408	0.977944236
25	212.2515608	0.208157192	0.001065195	0.314019929	4	ReLU	5	0.15574223	8	24	2	4	Conv	3	FALSE	0.329539786	2	40	0.957575758	0.980024969	0.977944236
327	211.5136221	0.257221665	0.001712906	0.291625457	2	RReLU	5	0.138509645	8	10	1	3	Conv	3	FALSE	0.46974108	2	63	0.957843137	0.979608822	0.978360383
66	108.605852	0.162372259	0.031561904	0.562044063	4	ReLU	5	0.102112963	10	14	2	4	Ave	2	TRUE	0.469361396	1	40	0.953208556	0.979608822	0.975031211
122	130.2673093	0.087374794	0.000419329	0.390607716	2	ELU	5	0.193262501	10	21	2	3	Max	3	TRUE	0.479510263	2	104	0.940909091	0.97752809	0.978360383
263	215.2265518	0.458219489	0.371604462	0.732216952	3	PReLU	3	0.14953052	8	23	2	2	Conv	2	TRUE	0.360438715	2	37	0.967112299	0.977111943	0.978776529
576	163.5379525	0.105631985	0.051693551	0.235841211	3	PReLU	5	0.109424895	9	28	2	2	Max	2	TRUE	0.389384822	2	72	0.953743316	0.97627965	0.970869746
138	235.7162382	0.207166371	0.000534136	0.48111136	2	PReLU	3	0.151626466	9	25	2	4	Conv	3	FALSE	0.412929841	2	33	0.953654189	0.975863504	0.974615065
310	158.2980798	0.074132803	0.013175034	0.61744286	3	PReLU	5	0.124480356	10	13	2	4	Conv	3	FALSE	0.334739045	1	100	0.959447415	0.975447357	0.974198918
53	190.5642876	0.178850727	0.14274637	0.071829751	1	RReLU	5	0.133074498	12	19	1	2	Max	3	TRUE	0.441818465	1	52	0.953832442	0.974198918	0.972534332
97	143.8217161	0.175332387	0.061911864	0.004233371	1	ReLU	3	0.109264426	8	28	1	3	Conv	3	TRUE	0.345839556	1	80	0.971568627	0.973782772	0.975447357
199	142.6118065	0.112825226	0.086458226	0.761143231	2	ELU	5	0.123948145	8	16	1	2	Max	3	FALSE	0.466729135	2	15	0.94741533	0.973782772	0.974615065
298	56.03592741	0.150514536	0.05969678	0.635792658	4	RReLU	5	0.194913645	11	28	1	3	Max	3	FALSE	0.326384701	2	66	0.949910873	0.973782772	0.976695797
99	122.5566254	0.116638341	0.003609139	0.664429483	3	RReLU	5	0.110248306	8	29	2	4	Ave	2	TRUE	0.470381987	2	93	0.952406417	0.973366625	0.975863504
116	207.7327939	0.464397431	0.107600328	0.611855376	1	ReLU	5	0.142859392	10	22	1	2	Conv	3	TRUE	0.349230422	2	53	0.97486631	0.972950479	0.972534332
245	173.4445489	0.289095757	0.054145047	0.576235865	4	PReLU	5	0.189666114	12	27	2	3	Ave	3	TRUE	0.387981265	1	76	0.940819964	0.972534332	0.9704536
362	81.81765475	0.380502267	0.001057016	0.708985295	2	PReLU	3	0.113998241	12	24	2	4	Conv	3	TRUE	0.466903861	1	32	0.957308378	0.972534332	0.970037453
635	250.8837523	0.067567463	0.051450302	0.869826144	1	RReLU	5	0.162469737	11	28	2	2	Conv	3	FALSE	0.478596511	2	111	0.94714795	0.972118186	0.970037453
152	236.331373	0.222652376	0.0039135	0.360496498	3	ReLU	3	0.148694847	10	11	1	4	Max	2	FALSE	0.451867782	1	107	0.929322638	0.972118186	0.965459842
609	134.01304	0.105580211	0.014088728	0.088693959	4	ELU	3	0.175957068	9	20	1	4	Max	3	TRUE	0.389719522	2	101	0.939661319	0.972118186	0.972950479
393	172.1775567	0.064912833	0.000712177	0.777005928	4	ELU	3	0.155016493	10	16	2	3	Max	3	TRUE	0.432904238	1	12	0.932442068	0.972118186	0.969621307
477	139.8377391	0.429266626	0.040251979	0.325038814	1	ELU	3	0.194167878	11	26	2	3	Max	3	FALSE	0.326736593	1	44	0.937789661	0.971702039	0.96920516
596	87.73605288	0.104882569	0.006181763	0.738889037	2	ELU	5	0.185285173	9	11	2	3	Max	3	FALSE	0.367404176	1	117	0.931818182	0.971702039	0.970869746
172	119.6873832	0.560468759	0.005042358	0.677621188	4	ReLU	5	0.194403399	8	18	2	4	Conv	2	FALSE	0.318494145	1	54	0.950445633	0.971702039	0.970037453
69	159.4400251	0.282078601	0.014087521	0.519463051	4	RReLU	3	0.168037515	12	26	2	3	Conv	2	FALSE	0.306573409	1	75	0.946345811	0.971285893	0.967540574
297	182.8877851	0.25020883	0.000355533	0.701381009	4	ReLU	3	0.152464171	10	16	1	4	Max	3	FALSE	0.300551798	2	55	0.921746881	0.971285893	0.968372867
564	105.5165205	0.11688184	0.000366457	0.773670164	4	ELU	5	0.191220674	9	26	2	4	Max	2	TRUE	0.354802549	1	33	0.944741533	0.970869746	0.973782772
543	216.6339499	0.150274257	0.016655139	0.069461127	4	RReLU	3	0.17799817	12	28	2	3	Conv	2	TRUE	0.300268529	1	27	0.951693405	0.970869746	0.972534332
267	72.16311724	0.87726686	0.026220397	0.630886329	4	PReLU	5	0.151800717	8	15	2	2	Ave	3	TRUE	0.441068643	2	18	0.931729055	0.970869746	0.966708281
204	271.0151177	0.086754221	0.026320428	0.760668932	3	ReLU	5	0.171142847	9	18	1	3	Conv	3	FALSE	0.400276817	2	118	0.948039216	0.9704536	0.972118186
430	198.0002623	0.416932564	0.001795343	0.532197142	1	ELU	3	0.123151676	12	30	2	4	Ave	3	TRUE	0.321682336	1	67	0.958823529	0.970037453	0.970869746
544	116.0911452	0.092938733	0.01624934	0.787286902	3	PReLU	5	0.166722884	8	30	2	2	Conv	2	FALSE	0.401841022	2	119	0.949910873	0.969621307	0.964627549

Table B.2. Results with extra crops

BIBLIOGRAPHY

Abdolvahabi, A., B. W. Taylor, R. L. Holden, E. V. Shaw, A. Kentsis, C. Rodriguez-Galindo, S. Mukai, and B. F. Shaw (2013). Colorimetric and longitudinal analysis of leukocoria in recreational photographs of children with retinoblastoma. *PloS one 8*(10), e76677.

Abramson, D. H. and A. C. Schefler (2004). Update on retinoblastoma. *Retina 24*(6), 828–848.

Cibur, V. (2016). Facetag, the image managing service for the leukocoria detection project.

Clevert, D.-A., T. Unterthiner, and S. Hochreiter (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*.

Graham, B. (2014). Fractional max-pooling. *arXiv preprint arXiv:1412.6071*.

He, K., X. Zhang, S. Ren, and J. Sun (2015a). Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*.

He, K., X. Zhang, S. Ren, and J. Sun (2015b). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034.

Henning, R., P. Rivas-Perea, B. Shaw, and G. Hamerly (2014). A convolutional neural network approach for classifying leukocoria. In *Image Analysis and Interpretation (SSIAI), 2014 IEEE Southwest Symposium On*, pp. 9–12. IEEE.

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Karpathy, A. and J. Johnson (2016). Stanford cs231n: Convolution neural networks for visual recognition.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105.

Lee, C.-Y., P. W. Gallagher, and Z. Tu (2015). Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. *arXiv preprint arXiv:1509.08985*.

Maas, A. L., A. Y. Hannun, and A. Y. Ng (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, Volume 30, pp. 1.

Mishkin, D. and J. Matas (2015). All you need is a good init. *arXiv preprint arXiv:1511.06422*.

Ries, L. A. G., M. A. Smith, J. Gurney, M. Linet, T. Tamra, J. Young, G. Bunin, et al. (1999). Cancer incidence and survival among children and adolescents: United states seer program 1975-1995. *Cancer incidence and survival among children and adolescents: United States SEER Program 1975-1995*.

Springenberg, J. T., A. Dosovitskiy, T. Brox, and M. Riedmiller (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15*(1), 1929–1958.

Xu, B., N. Wang, T. Chen, and M. Li (2015). Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*.

Yu, F. and V. Koltun (2015). Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*.