Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Jani Kettunen

Bayesian Hyperparameter Optimization of Gaze Estimation Neural Networks

Master's Thesis
Espoo, November 2, 2016

Supervisors: Professori Aki Vehtari, Aalto University
Advisor: Professor Aki Vehtari

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Jani Kettunen

Title:
Bayesian Hyperparameter Optimization of Gaze Estimation Neural Networks

Date: November 2, 2016 **Pages:** 58

Major: Computer Science **Code:** T-110

Supervisors: Professor Aki Vehtari

Advisor: Professor Aki Vehtari

Neural networks have seen a surge in usage during the last decade. They involve several hyperparameters which need to be tuned in order to achieve the best performance. Lately Bayesian hyperparameter optimization methods have shown good results on this task.

Gaze estimation has traditionally been done in controlled laboratory environments or with equipment not widely available. Lately there have been efforts to bring gaze estimation closer to the real world by creating datasets with realistic lighting and by using commercially mainstream cameras. Neural networks have previously been applied to these datasets, but the network structure has been left unoptimized.

This thesis utilizes Bayesian hyperparameter optimization to improve the neural networks previously used for gaze estimation.

The results show that Bayesian hyperparameter optimization does improve the previous results, and we improve on the previous best by 3 percent. However, we also show that Bayesian hyperparameter optimization left room for improvement by utilizing neural network ensembles to achieve a 6 percent improvement. More generally, we conclude that utilizing Bayesian hyperparameter optimization is a relatively easy way to increase network performance, but comes with its own caveats.

Keywords: hyperparameter optimization, neural networks, dropout, gaze estimation

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Jani Kettunen		
Työn nimi:	Bayesialainen hyperparametrioptimointi katseen suuntaa arvioiville neuroverkoille		
Päiväys:	2. marraskuuta 2016	**Sivumäärä:**	58
Pääaine:	Tietotekniikka	**Koodi:**	T-110
Valvojat:	Professori Aki Vehtari		
Ohjaaja:	Professori Aki Vehtari		

Neuroverkkojen suosio on lisääntynyt viimeisen vuosikymmenen aikana. Jotta niitä voitaisiin käyttää tehokkaasti ja saavuttaa hyviä tuloksia, täytyy määritellä useita niin sanottuja hyperparametreja. Viime aikoina Bayesialaisilla menetelmillä on saatu hyviä tuloksia neuroverkkojen hyperparametrien optimoinnissa.

Katseen suunnan tunnistus on perinteisesti tehty laboratorio-olosuhteissa tai käyttäen laitteistoa, jota ei ole laajalti saatavilla. Viime aikoina on kuitenkin julkaistu tutkimuksia liittyen katseen suunnan tunnistukseen realistisessa valaistuksessa ja käyttäen laajalti käytössä olevia kameroita. Näiden tuloksien saavuttamisessa on hyödynnetty myös neuroverkkoja, mutta neuroverkkojen hyperparametreja ei ole erikseen optimoitu.

Tässä diplomityössä tutkitaan Bayesialaisen menetelmien käyttöä neuroverkkojen hyperparametrien optimointiin ja sovelletaan sitä katseen suunnan tunnistukseen.

Tulokset osoittavat, että Bayesialainen hyperparametrioptimointi paransi tuloksia ja paransimme parasta tämänhetkistä tulosta 3 prosentilla. Näytämme kuitenkin myös, että hyperparametrioptimointi jätti parannusvaraa käyttämällä neuroverkko-ensemblea joka paransi tuloksia 6 prosentilla. Yleisemmällä tasolla havaitsimme, että Bayesialainen hyperparametrioptimointi on suhteellisen helppo tapa parantaa verkon suorituskykyä, mutta parhaiden tulosten saaminen vaatii myös ihmisen asiantuntemusta.

Asiasanat:	hyperparametrioptimointi, neuroverkot, katseen suunnan tunnistus
Kieli:	Englanti

Abbreviations and Acronyms

GPU	graphics processing unit
RAM	random access memory
GB	gigabyte
GHz	gigahertz
MPII	max planck institut informatik
SMAC	sequential model-based algorithm configuration

Contents

Chapter 1

Introduction

Since 2006 neural networks have seen a surge in research interest and usage. This has been partially due to state of the art results in computer vision tasks, such as object recognition. [8]

Gaze estimation has a long history due to the number of applications it has. It has been used to keep track of drivers getting tired, examine usage patterns of internet sites, and recently there have been products aimed at complementing or even replacing mouse usage with gaze tracking. In the past gaze estimation was usually done using external hardware, like infrared lights or head-mounted gear. After computer vision methods started to gain popularity, gaze estimation moved towards analyzing the gaze direction directly from images and videos. For a long time these methods were based on handcrafted features extracted from the images, but recent years have seen a move towards neural networks and feature learning. [38] [39]

Neural networks require many parameters to be set prior to training them, called hyperparameters. Those are often set manually, but automated methods are also available. However, it's not unusual for large neural networks to take several days to train, and the training usually requires powerful hardware. This means that frugality with regards to the number of training runs is important.

The above problem can be modelled as an optimization problem of an unknown function with some given constraints. For this type of a problem, Bayesian optimization methods are among the most efficient ones in terms of function evaluations [11]. This is why Bayesian optimization of neural network hyperparameters has gained popularity recently and lead to some state of the art results in competitive datasets [6].

Problem statement and goals

Since cameras are becoming more and more ubiquitous, it makes sense to see how well gaze estimation can work without more customized hardware like infrared lights. This has a lot of potential commercial applications and can work in places where using additional hardware isn't feasible, for example in outdoor spaces. In addition, solutions based on high resolution cameras are likely to be cheaper than solutions based on custom hardware.

Some studies have applied neural networks to gaze estimation tasks, but so far none of them have applied systematic methods to optimize the hyperparameters of these neural networks. This suggests there might be some room for improvement by utilizing those methods. In addition, it's interesting to see how well current Bayesian hyperparameter optimization methods perform.

As such, the goals of this thesis are twofold: to examine Bayesian hyperparameter optimization of neural networks and to apply it to the task of gaze estimation. Also, we're interested in finding out if systematic hyperparameter optimization can improve on the previous unoptimized results, and if it can, what's the margin for improvement.

Structure of the thesis

Chapter 2 introduces the basics of neural networks. The goal of this chapter is to introduce the common hyperparameters that are tuned when optimizing neural networks.

Chapter 3 talks about the field of hyperparameter optimization and the currently most popular methods. In addition, we'll take a deeper look at Bayesian optimization from the perspective of neural network hyperparameter optimization.

In chapter 4 we outline the field of gaze estimation and the most popular datasets which are available for our task. In addition, we'll introduce the most common approaches taken with these datasets.

Chapter 5 describes the implementation details of our experiments. We describe the starting point we have, the tools we've chosen and the methods applied.

Chapter 6 introduces the results and compares them against the previous best.

Chapter 7 discusses the lessons learned, conclusions we can draw and outlines some possible future improvements.

Chapter 8 presents a summary of our findings.

Chapter 2

Neural networks

In this chapter we'll introduce neural networks, explain how they are trained and review common regularization methods used to help in training them. The purpose is to give an overview of a typical neural network training process and the related terminology, in order to later talk about the parameters needed during that process.

Feedforward neural networks

There are two major categories of neural networks: feedforward and recurrent. We're going to concentrate on feedforward neural networks, since they are the most used network type in computer vision and will be used by us in the implementation part.

A feedforward network receives a real-valued input vector x and, by performing a series of computations on x, it produces a real-valued output vector y. The exact nature of the computations is defined by parameters θ. Thus, a feedforward network can be defined as a function $y = f(x; \theta)$. [7]

When the computations can be represented as a directed acyclic graph, the network is called a feedforward network. This is in contrast to recurrent networks, where the graph can have cycles. This would mean that a result of a computation can affect the result of a later computation from that same unit. [7]

A feedforward neural network is typically composed of layers. Let's say the first layer, given input vector x, creates output $f^{(1)}(x)$ and hands its output to the second layer which, in turn, creates output $f^{(2)}(f^{(1)}(x))$. This would correspond to a feedforward neural network with three layers (the two mentioned, plus input layer) and an output of $y = f^{(2)}(f^{(1)}(x))$. The first layer is often called the input layer, the last layer the output layer, and the

layers between them the hidden layers. [7]

Next, we'll show how a typical layer in a feedforward network can be modelled. The output vector y of the layer is fully defined by its input vector x, weight matrix w, bias vector b and activation function σ according to the following formula: $y = \sigma(wx + b)$ where $\sigma(.)$ denotes the pointwise application of function σ to each element of its input vector. Now, when these layers are stacked on top of each other and we denote the parameters of n'th layer with superscript ^{n}, we get $x^{n+1} = \sigma(w^n x^n + b^n)$. Often the activation function is the same for the whole network, which is why we omitted the superscript with σ. Common choices for σ are the rectifier (2.1) and the logistic function (2.2). The length of vector x^{n+1} can be referred to as the width of the n'th layer. [26]

$$\sigma(x) = max(x, 0) \tag{2.1}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

From above we can note that a single output value of a layer can be formed by multiplying each output of the previous layer with a corresponding weight, summing the formed values and the bias term, and by applying an activation function to the resulting sum. This process is often said to be loosely inspired by the neurons in the brain, which is why neural networks are often said to be composed of neurons. Neurons might also be referred to as units, and if we're specifically talking about the neurons in input, hidden or output layers, we can talk about input units, hidden units and output units, correspondingly. Figure 5.2 shows an example of a neural network with one layer of each type. The edges represent the connections between different layers. One edge can be thought of as corresponding to a single element of the weight matrix w.

The weight matrices, bias vectors and the activation function (or functions) of the network form the aforementioned parameters θ. When applying feedforward networks, we want the parameters θ to have such values that the function f approximates some target function f^* as well as possible. To find out how well f approximates f^*, we need a function which measures how different the output $f(x)$ is from the target output $f^*(x)$. This function is called the cost function. An example of a common cost function is mean squared error.

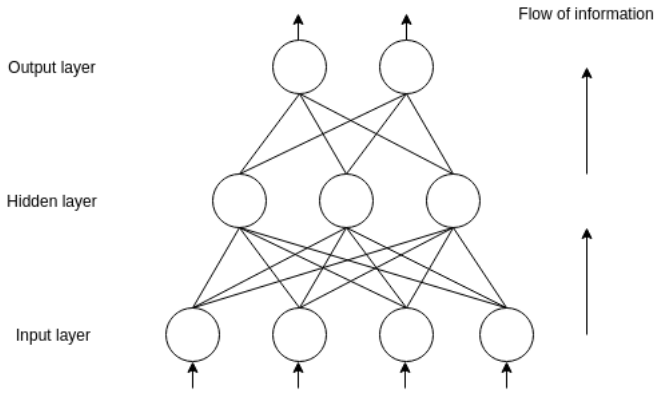

Figure 2.1: An example of a typical neural network with one hidden layer.

Backpropagation and optimization

So far we've presented an example of a typical neural network but haven't explained how exactly we plan to set its parameters. A common way to initialize the parameters is at random. There are several ways to do this, but here we'll skip the details. Instead, we'll focus on what happens next. Without further modification of the parameters, the neural network would forever be stuck outputting more or less random values for any given input. To fix this, we train the neural network.

Backpropagation is a method for computing the gradient of the loss function with respect to any weight in the network and is an essential part of the training procedure of modern neural networks. It's paired with some optimization method to modify the weights of the network so that the output $f(x)$ will take steps closer to the target value $f^*(x)$. In this section we'll present backpropagation closely following [24] and [26] and talk about the optimizers.

As seen in previous chapter, the output of the whole network can be expressed as

$$x^{f+1} = \sigma(w^f x^f + b^f)$$

where superscript ^{f} denotes final layer. For simplicity, let's define $y = x^{f+1}$. We'll denote the n'th member of vector x with x_n and the element at the i'th row and j'th column of matrix w with $w_{i,j}$. Now we can write [26]:

$$y = \sigma(\sum_k w^f_{n,k} x^f_k + b^f_n)$$

If we choose a network with mean squared error as the cost function we can write the error of a single output unit as

$$E_n = \frac{1}{2}(y_n - x_n^*)^2$$

and the total error as

$$E_{total} = \frac{1}{2}\sum_n (y_n - x_n^*)^2$$

where x^* is the target output vector.

Next, we'll consider the effect of single element of w^f, denoted $w_{i,j}^f$. We'll define

$$net_n = \sum_k w_{n,k}^f x_k^f + b_n^f$$

and, as previously, $y_n = \sigma(net_n)$.

Specifically, we want to find out how much the total error changes when the value $w_{i,j}^f$ is changed. In mathematical terms, we want to find out the partial derivative of the total error with respect to a change in one of the weights. By using the chain rule, we get

$$\frac{\partial E_{total}}{\partial w_{i,j}^f} = \frac{\partial E_{total}}{\partial y_n}\frac{\partial y_n}{\partial net_n}\frac{\partial net_n}{\partial w_{i,j}^f} \tag{2.3}$$

Now we can calculate each piece separately. With mean squared error as the cost function, we have

$$\frac{\partial E_{total}}{\partial y_n} = y_n - x_n^*$$

Also, since we chose cost function σ to be the logistic function, we have

$$\frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1 - \sigma(x)) \tag{2.4}$$

which leads to

$$\frac{\partial y_n}{\partial net_n} = \frac{\partial}{\partial y_n}\sigma(\sum_i y_i) = \frac{\partial}{\partial y_n}\sigma(y_n) = \sigma(y_n)(1 - \sigma(y_n)). \tag{2.5}$$

Also, it's fairly straightforward to see that

$$\frac{\partial net_n}{\partial w_{i,j}^f} = w_{i,j}^f \tag{2.6}$$

Now when we plug in the results from 2.4, 2.5 and 2.6 to 2.3 we get the final result

$$\frac{E_{total}}{\partial w_{i,j}^{f}} = (y_n - x_n^*)(\sigma(y_n)(1 - \sigma(y_n))w_{i,j}^{f} \tag{2.7}$$

where σ is the logistic function. We can repeat this process for bias values. The above process can also be repeated for further layers by extending the chain rule to calculate the gradient for the elements in those layers.

Now we have calculated the gradient of the total error with respect to a single weight. The next step is to use this information to modify the weights to reduce the total error.

To decrease this error, we subtract the resulting value from $w_{i,j}^{f}$. Optionally, we can multiply it with some value before substracting it, that is, instead of subtracting the result given by 2.7 we subtract it multiplied by some value α, where α is typically between 0 and 1 and is called the learning rate.

There are several algorithms for deciding how to modify the learning rate while the training is in progress to get the best result [28]. These are commonly referred to as optimizers [8].

Convolutional neural networks

Convolutional neural networks are a special type of feedforward neural network which utilize the convolution operation.

Discrete convolution of real-valued functions f and g is given by: [14]

$$(f * g)[n] = \sum_{m=-\infty}^{m=\infty} f[m]g[n - m] \tag{2.8}$$

In the terminology of convolutional neural networks, the first argument (f above) is often referred to as the input and the second argument (g) as the kernel. The output can be referred to as a feature map. A typical convolutional layer can have several feature maps, each produced by a convolution with a separate kernel. A kernel is somewhat analogous to the weight matrix of the neural network seen in section 2.1, but is typically more localized.

Figure 2.2 shows an example of a one-dimensional convolutional layer with two feature maps. The kernel size is three, as each unit in the convolutional layer takes inputs from three neurons. In addition, all units in a feature maps share the same kernel. This is represented by the colored edges: the edges which share their color also share their weight.

While equation 2.8 shows the formula for one-dimensional convolution, with neural networks it's also common to convolve over several dimensions. For example two dimensional convolution is common when the inputs are images. [7]

In the previous section we had a model where each output from layer n was connected to each input in layer $n + 1$. In this case layer $n + 1$ is often described as fully connected. [26] However, this is not the case with convolutional neural networks if the kernel is smaller than the input, which can also be seen in figure 2.2 where the leftmost units of the feature map aren't affected by the rightmost input units. [7]

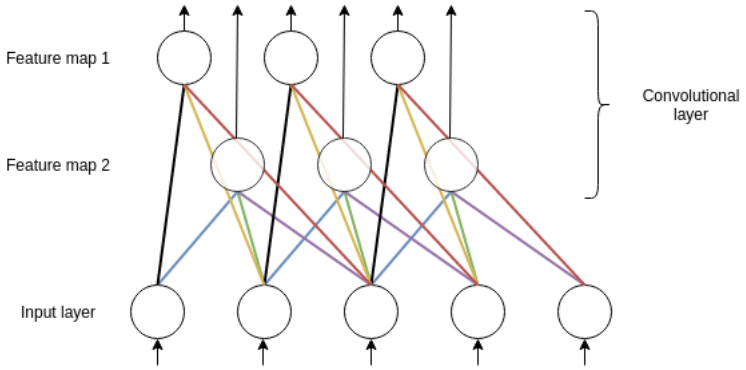

Figure 2.2: An example of a convolutional layer. Edges with same colors have the same weights.

Often convolution is followed by pooling, which replaces the output of the layer at certain locations with summary statistics. For example, max-pooling replaces the output of neighboring units with their maximum value. [7]

Regularization

When training neural networks, we usually have separate data for training and testing the algorithm. This is to give us a better idea of how the algorithm performs on unseen data. When training a neural network it's common to see its performance improve in the training set but decrease in the test set. This means that the algorithm is simply learning the quirks of the training set. This is called overfitting.

There are several strategies for making the algorithms prevent overfitting, and they're commonly called regularization. Regularization can be defined as "any modification we make to a learning algorithm that is intended to

reduce its generalization error but not its training error". [7] In this section we'll present some of the most common regularization strategies.

Here we introduce some of the most common regularization strategies that modern neural networks use. Our aim is to present the techniques and how they relate to each other. The techniques we'll present here are the ones we'll use in the implementation of our network, or refer to when planning the details of the implementation.

Dropout is a fairly recent technique which randomly omits some of the units of neural network during training. This prevents the units from co-adapting too much and causes the network to learn several independent representations. The rate at which the units are dropped is referred to as the dropout rate [34] [7]. This relatively simple technique is nowadays part of the training process in many, if not most state-of-the-art object detection networks [6].

The effect of dropout is the same as using the mean output from many separately trained networks, with the aim of having the errors cancel each other out. [34] This approach can be referred to as using an ensemble of neural networks.

L^2 parameter regularization, commonly known as weight decay, adds a regularization term to the total error of the network. This regularization term is the sum of all the squared weights: $\frac{\lambda}{2}\sum_i w_i^2$ where $\lambda > 0$ is called the regularization parameter. Note that the biases are excluded. The effect of L^2 regularization is to create a network which favors smaller weights. [8] [26]

Another regularization technique is early stopping, which keeps track of the of-out-sample error and stops when some criterion is fulfilled, for example if the error hasn't improved during the last few epochs. One way to achieve this is to divide the data into training, validation and test sets. We can then train the model with the training data and use the validation set to check when the error stops improving. Finally, we use the test set to get the actual error. The separation between validation and test data is important: if we made a decision on which model to choose based on the results obtained with the test set, our error wouldn't be based on unseen data.

There is a connection between L^2 regularization and early stopping: they essentially play the same role. [8]

As can be seen above, regularization methods often rely on restricting the values the units in a neural network can take. An interesting question that arises from this is why use regularization at all? Couldn't we simply make the network smaller, in terms of width and number of layers? Often it's simply more convenient: instead of modifying the size of each layer we just modify a single parameter which might have an effect on all the layers.

However, a more compelling reason is that empirically, it's been shown that for large networks we will be much more likely to end up with a high quality local minima during training, as opposed to small networks where the local minima is more likely to be of low quality [13]. As such, it'd make sense to focus on larger networks and use regularization instead of trying to make the network smaller.

Chapter 3

Hyperparameter optimization of neural networks

We can define hyperparameter as a variable "to be set prior to the actual application of the learning algorithm to the data, one that is not directly selected by the learning algorithm itself." [8]

As was seen in chapter 2, neural networks have a large amount of parameters which are typically trained using a combination of backpropagation and some optimization method. However, there are some parameters which can't be set using that process. These parameters are typically the hyperparameters of the neural network, and we need to set these parameters some other way.

Intuitively, we want to find out what are the hyperparameter values with which we get the best performance out of the network. Unfortunately, this process is often considered to be a "black art" in the sense that it's hard to formalize [31].

Fortunately, several methods have been invented in an attempt to formalize or automate this process, coined hyperparameter optimization. This chapter will cover hyperparameter optimization of neural networks. We'll cover the most common methods used to optimize neural network hyperparameters and especially focus on Bayesian hyperparameter optimization. To properly present Bayesian hyperparameter optimization, we'll also cover some of the mathematics needed to understand the methods.

Neural network hyperparameters

Neural networks can include a large number of hyperparameters, as seen in chapter 2, for example learning rate, size and depth of the layers and

dropout rate. For convolutional neural networks, we have hyperparameters such as kernel size and the number of feature maps in convolutional layers. Even some activation functions might require their own hyperparameters, for example some variants derived from the rectified linear unit, although in this thesis we'll only work with standard rectified linear units [37].

We can broadly categorize hyperparameters to two classes. For one, we have numerical hyperparameters, which are integers or real numbers [8]. They're also usually locally correlated, that is, two parameters with a small difference between them correlate more strongly than two parameters with a large difference. For example, learning rate and the number of units in a layer belong to this category. In addition, we have categorical hyperparameters [18]. For example the activation function type and the optimizer algorithm to use belong to this category.

In addition to numerical and categorical hyperparameters, we can define some hyperparameters as conditional [18]. These hyperparameters depend on some other hyperparameter. For example, in order to define the width of each layer, we need to define the number of layers first, making the width of a layer a conditional hyperparameter.

Practical considerations

When selecting the approach to hyperparameter optimization, we have some practical considerations which limit and define our choices.

A major consideration is our prior knowledge of the distribution. For example, for several hyperparameters it makes more sense to optimize them in the logarithmic domain. An example is the initial learning rate: if the learning rate is 0.1, a change of 0.001 will have a much smaller impact than a similar change if the learning rate was 0.002. [8] This knowledge can be utilized when searching for the best hyperparameters, or alternatively we need to make sure this knowledge can be somehow derived by the hyperparameter optimizer.

Another thing to consider are resources, be it human, time or computational. The computational resources we can use mainly affect the number of iterations we can run, both sequentially and in parallel. This affects the number of hyperparameters we can optimize. There also are strategies to make better use of computational resources, for example using a computationally cheap estimator to calculate the validation error. [8] On the negative side, this would increase the complexity of the code required, requiring more human resources.

Third consideration is related to choosing the hyperparameters to opti-

mize. If computational resources won't allow for efficient search of all hyperparameters, we're forced to reduce the search space by limiting our search to a subset of the hyperparameters. In practise it's usually only a few hyperparameters that matter, so when properly selected, this limitation shouldn't have a large impact on the results [18] [8]. However, this requires prior knowledge of the important hyperparameters or at least a way to determine them.

In addition to the above, reproducibility of the optimization process is often of major interest.

Non-Bayesian hyperparameter optimization

When optimizing hyperparameters we want to find such a set of hyperparameter values that after being trained, the network minimizes its error over all hyperparameter configurations. There are several common strategies for achieving this. These strategies are generally applicable to any machine learning algorithm with hyperparameters.

The most straightforward strategy is manual search. In manual search the user or users try to identify promising hyperparameter regions and develop intuition for selecting hyperparameters [9]. The obvious downsides of this strategy are reproducibility and the human effort required.

Another fairly simple strategy is grid search. In grid search we choose a set of possible values for each hyperparameter and test every possible combination. This means that the number of combinations to evaluate is exponential compared to the number of hyperparameters. With grid search it's typical to try several value ranges with one being more localized than the previous one, meaning that often grid search needs to be combined with manual search to narrow down on the exact values of the optimal parameters. [8] This is because in practise, grid search alone does very poorly [9]. On the other hand, compared to manual search, grid search has the upside of being better able to exploit parallerism. [8]

Compared to the above mentioned, a more efficient approach is random search. In random search, we choose a prior distribution over the allowed values for each hyperparameter. The distribution can be continuous or discrete depending on the type of the parameter. We then draw each hyperparameter value from their respective distributions for each new evaluation. One benefit of this strategy is that we can easily encode our prior knowledge to the distributions, and at worst the distributions can simply be uniform. In addition, random search can also exploit parallelization. [8] In practise random search is much more efficient than grid search, especially as the number

of parameters increases. [10]

There are several other approaches in addition to the ones mentioned. However, of particular interest to us is Bayesian optimization of neural network hyperparameters, which we'll present next in more detail.

Bayesian hyperparameter optimization

Bayesian optimization of neural network hyperparameters has gained popularity recently, partially due to good results in competitive datasets such as CIFAR [6] [33]. In this section we'll briefly explain the mathematics needed to understand the methods and talk about their applications.

Gaussian processes and regression

A common element in Bayesian optimization of neural network hyperparameters is the Gaussian process and many successful hyperparameter optimization methods utilize it in some way [31] [22].

Formally, we can define Gaussian process as a collection of random variables, so that each finite collection of those random variables has multivariate normal distribution.

A Gaussian process is completely specified by two function: the function $m(x)$ which gives the expected mean for vector x, and $k(x, x')$ which gives the expected covariance between arbitrary vectors x and x'. In our case, the vector x would represent the neural network hyperparameters we have selected.

Following the above notation we can write

$$f(x) \sim GP(m(x), k(x, x'))$$

where $f(x)$ is, in our case, the distribution of the error for our network, given a vector of hyperparameters x.

For calculating the covariance $k(x, x')$ there are several well-known functions. A commonly used covariance function is the squared exponential function:

$$k(x, x') = exp(-\frac{1}{2}|x - x'|^2) \tag{3.1}$$

Next, we'll define D as the data we have, that is, our observations and their cost. In our case that would mean the hyperparameter settings we have evaluated and their corresponding errors. Now we can make predictions from the data using Bayes' formula:

$$p(f|D) = \frac{p(D|f)p(f)}{p(D)}.$$

More specifically, we can condition the posterior on the observations we have made. Let's assume we have observations from some points, the points being represented by X, and we want to predict the mean and variance at some arbitrary points X'. Since we have defined the covariance function k, we can calculate the covariance matrix $k(X', X)$ which includes the covariances between each observation in X and X' (but, in particular, excludes the covariances between the elements within X or X'). Similarly, we can denote by $k(X, X)$ the covariance matrix between each element in X and similarly for $k(X', X')$. In addition, let's assume our observations are noisy with a variance of σ_n^2. Now the expected mean for points X' is given by

$$k(X', X)[k(X, X) + \sigma_n^2 I]^{-1} y$$

where vector y is the noisy observations and I is the identity matrix. The covariance matrix for X' is given by

$$k(X', X') - k(X', X)[k(X, X) + \sigma_n^2 I]^{-1} k(X', X').$$

[27] Here we'll skip the derivation of these formulas. However, it's worth noting that making predictions involves matrix products and inversions, making its complexity $\mathcal{O}(n^3)$ with naive algorithms, where n is the number of observations so far.

Applying Bayesian hyperparameter optimization with Gaussian processes

Bayesian optimization techniques can be used to find the extrema of an objective function. They're especially useful in situations where the function is non-convex, have no derivatives and when the evaluation is costly, since Bayesian optimization techniques require relatively few function evaluations. [11]

This section will give a high-level overview of how Gaussian processes can be used with Bayesian optimization to optimize neural network hyperparameters. We will see this process in action in the implementation part.

Intuitively, from the perspective of neural net hyperparameter optimization, a Gaussian process conditioned on previous observations tells us how well the hyperparameter combinations we have observed so far correlate with the unknown, not yet observed combinations. This gives us a way to decide which combinations we want to explore next.

First we establish the Gaussian process prior by defining the mean and covariance funtions. After that, we gather data by selecting new points to evaluate, the selection being based on the information provided by our prior and previously evaluated data points.

Typically each observation improves our understanding of the objective function in all the input space. An example of this can be seen in figure 3.1 where we want optimize over a single variable x. The black line is the expected mean, the grey area represents the variance. Initially, we only have the prior distribution (left) to rely on. After one function evaluation at $x = -2$ (center) we get a better idea of the expected mean and variance around that point, and after two function evaluations (right) we already have a fairly good idea of the shape of the function. In a real world scenario we'd typically optimize over more than one variable.

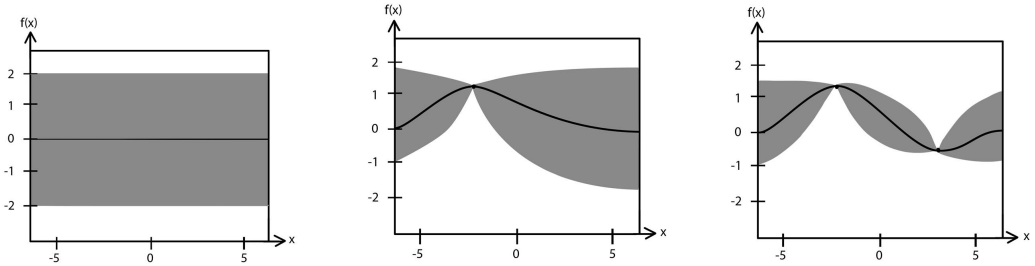

Figure 3.1: Posterior conditioned on zero, one and two data points.

An obvious consideration from above is how to exactly select the new points. We can't simply evaluate a set of random points since the purpose is to avoid the expensive evaluations as much as possible. For this end, we usually define an acquisition function, which can be evaluated cheaply in promising points of the posterior function. One example of a simple acquisition function is called probability of improvement:

$$PI(x) = P(f(x) \geq f(x^+))$$

$$= \Phi(\frac{\mu(x) - f(x^+)}{\sigma(x)})$$

where x^+ is the current best observation and $\Phi(.)$ is the normal cumulative distribution function.

The selection of acquisition function reflects how we want to balance exloitation and exploration. When favoring exploitation, the acquisition function is biased to select points with low mean, whereas favoring exploration means focusing on points with high variance. [11]

One thing to note from above is that the result from an evaluation won't improve our understanding of the hyperparameter space until after it's been evaluated. This means that the procedure doesn't lend itself to parallelization as naturally as random search. However, methods have still been developed to allow for parallelization [16].

Libraries

In this section we'll introduce some libraries we consider using in the implementation. We'll leave most of the implementation details out, but present a general overview and outline some results that have been achieved with the libraries. This list is not meant to be an exhaustive one, rather it's a list of the libraries we consider for the task.

Spearmint is a fairly well established library. It uses a Gaussian process to model the hyperparameter space. [31] In addition, after its initial publication it has had some interesting features added to it, such as built-in capability to recognize and handle variables which should be optimized in the logarithmic space. [32] [30] Due to its popularity its an interesting candidate to take into consideration.

Sequential model-based algorithm configuration (SMAC) is another fairly well established library. As opposed to Spearmint, it uses random forests to model the hyperparameter space.

GPyOpt [4] is a relatively new library. It mainly utilizes Gaussian processes to model the hyperparameter space, but also offers the option of using random forests. It has the same core functionality as Spearmint, plus some added features like visualization options and parallelization.

Fabolas is also a fairly new library with some impressive results. Its central idea is to first train on smaller subsets of data to get a rough idea of the optimal hyperparameter values, and gradually expand the data while honing in on even more optimal hyperparameter values. [22] However, using this library proved to be somewhat difficult due to its unstable python support.

Current developments in Bayesian hyperparameter optimization

Some of the most recent approaches rely on using only parts of the data to get to the general areas of the optimal hyperparameter values, and optimize from there with growing amounts of data [21]. The overall trend seems to be towards efficient usage of cheap approximative functions. However, many of these approaches haven't been implemented into stable libraries yet.

In addition, parallelization of Bayesian optimization methods is a fairly new area with a lot of potential use cases.

Some recent work suggests that Bayesian optimization methods provide only a slight edge over random search for machine learning algorithms [23]. However, contrary results have also been presented. In other tests Bayesian optimization methods were found to be similar to random search initially, but after the initial phase they find the optimum much more quickly. The time taken in finding the optimum was found to be around 10 times faster with Bayesian methods [22].

Chapter 4

Gaze estimation

Eye movement can provide a lot of information about person's thoughts, intentions and feelings and, as such, gaze estimation has a lot of real world application. For example, gaze tracking can be used to find out if a car driver is distracted and not looking at the road ahead [35].

Here we'll use "gaze estimation" to refer to finding out the angle of the gaze relative to the observer. In addition, we might refer to "gaze detection" which more typically refers to the process of detecting gaze directed at the observer [29]. "Gaze tracking", on the other hand, is more often used to refer to gaze estimation from videos [17]. In this thesis we're mostly interested in gaze estimation, that is, finding out where the eye is looking relative to the observer, based on a single image.

In this chapter we'll introduce gaze estimation and briefly talk about the most common computer vision methods for gaze estimation. After that, we'll introduce the largest datasets available for gaze detection tasks. Following these, we'll go through the results which have been achieved using these datasets, and the methods used.

Gaze estimation

Gaze estimation has a long history from even before modern computers and computer vision methods. Older methods used to be based on recorded videos which were analyzed manually. [38] When computer vision became more accessible, gaze estimation saw a move to methods based on computer vision and handcrafted features. Most recently, gaze estimation methods have seen a trend of neural network based approaches being used. [39].

One way to categorize gaze estimation methods is to divide them into two main categories: feature-based and appearance-based. In addition, we could

have some other categories, such as natural light methods, but we won't go deeper into those since they're either not that popular or relevant to us.

Feature-based methods rely on the characteristics of human eye to identify distinct features, such as reflections or the locations of eye corners and pupils. [17] These techniques often use some additional hardware. For example, it's common to use infrared lights to create reflections on to the eye, which are invisible to the human eye, but can be picked up by sensors [15]. Since the light source location is known, its reflection can be related to the location of the pupil and the angle between them estimated. The vast majority of gaze estimation methods are feature-based.

Appearance-based gaze estimation, on the other hand, relies on the appearance of the eye under natural light. These methods don't extract separate features, but treat the image as the input. [17]

In this work we'll be concentrating on appearance-based gaze estimation, since using neural network directly on the image data falls under this category.

Human eye

Human eye has a different amount of receptors in different parts of the eye. The fovea, located near the center of the retina, has a relatively high density of receptors and as such, is the area of the eye with the highest acuity. It covers roughly one degree field of view. Outside of fovea, the acuity drops to half or less. [19]

In gaze estimation we're usually interested in the fovea. However, as can be seen in figure 4.1, the fovea is located withing the eye socket, so its common to track the pupil or iris instead. Essentially, we're inferring the location of the fovea.

The above issues place natural limits on the accuracy of gaze estimation. In particular, since the fovea covers roughly one degree, it's difficult to get accuracies below that level with camera-based methods. This is consistent with previous results. [5] [25]

Practical considerations

When building a gaze estimation system, there are some best practises and considerations that need to be taken into account.

One particular thing to note is that no method has reported head pose invariance. This means that the exact head pose is important information for every gaze estimator, as it has a big effect on the appearance of the eye. Some approaches require head pose information as a separate feature,

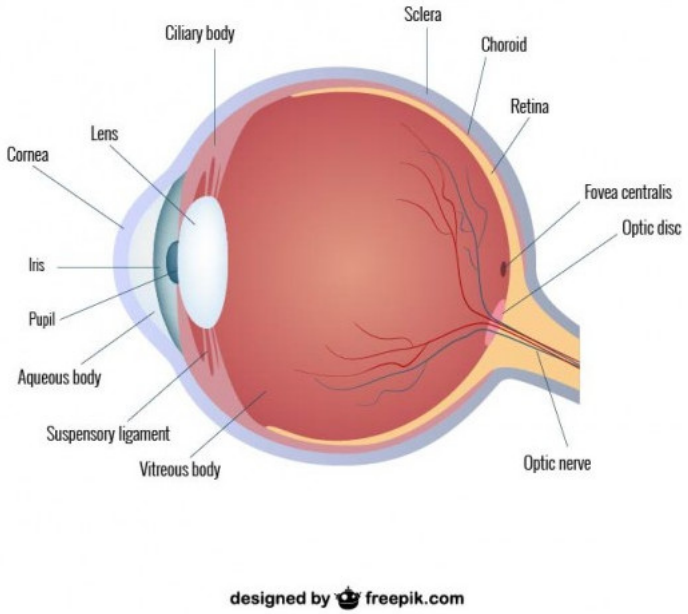

Figure 4.1: Structure of the human eye. [1]

and some bypass this requirement in some way, for example by warping and normalizing the images. [17][39]

In addition, we can do gaze estimation in a person-dependent or -independent way. Person-dependent gaze estimation means having the same people in the training and test set, and with person-independent gaze estimation there are no people from the training set in the test set. As could be expected, person-dependent gaze estimation has been shown to have much lower error rates. [39]

Datasets

This section will introduce the largest (at least 5000 images) gaze estimation datasets. In addition, some datasets are associated with some experimental data, which we're also going to present if available. These datasets differ in several qualities which affect their suitability for our task.

Dataset properties

When choosing a suitable dataset for our task, there are several properties we can use to differentiate between the datasets and prioritize them for our task.

For one, some datasets are completely or partially composed of 3D generated images. Using 3D images is a relatively easy and accurate way to expand the data set.

Second, the type of the images varies. Some datasets contain the whole face and a large part of the background, other datasets consist of images only consisting of the eyes or even one eye.

Third, the quality of the images varies. Some datasets have used high-resolution cameras and the pictures have been taken from a short distance, whereas others might have images from low-quality webcams.

Fourth, the variability of the images within a dataset differs significantly. Figure 4.3 and 4.2 show the difference in head and gaze angles between two of the largest datasets.

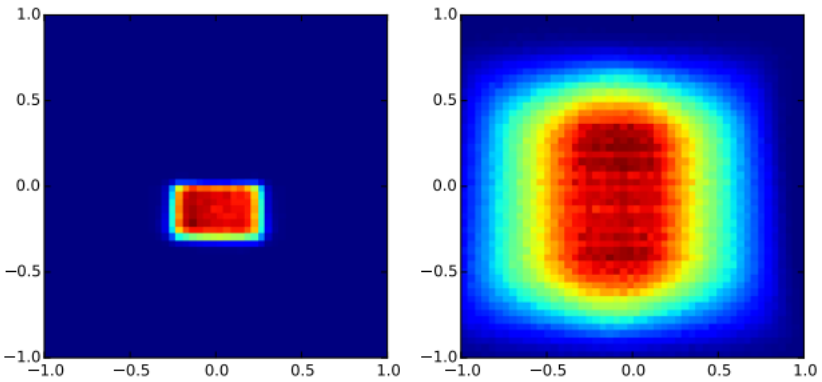

Figure 4.2: Distribution of gaze angle in MPIIGaze (left) and UT Multiview datasets. [39]

Fifth, the number of participants in the dataset is also a major consideration. With enough participants it's easier to estimate the accuracy of person-independent gaze estimation approaches, while with only a few participants the variance on the held-out data set would be high, making the estimation more difficult.

Finally, the datasets also differ in number of images. A higher number of images obviously helps with training machine learning algorithms.

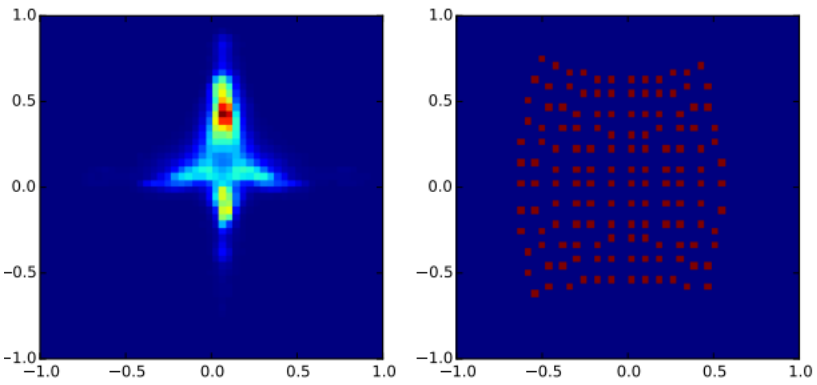

Figure 4.3: Distribution of head angle in MPIIGaze (left) and UT Multiview datasets. [39]

MPIIGaze

MPIIGaze dataset is a collection of 213,659 images. The paper where the dataself was published focuses on gaze detection 'in the wild', in the sense that the gaze detection is done in natural, everyday environments. The dataset is collected with webcams over a period of 46 days with the aim of introducing the kinds of images which could realistically be encountered in everyday life.

The data collection method was to install the image capturing software onto the laptops of 15 participants. Every 10 minutes the software automatically asked participants to look at a random sequence of 20 on-screen positions, visualized as a grey circle shrinking in size and with a white dot in the middle. The dataset is freely available.

The authors normalized the images before performing experiments. They extracted the eye region, changed the images from colored to black and white, warped them to make the eyes better face the camera and also histogram-equalised the intensity values. The cropped eye-region image and the final normalized image can be seen in figure 4.4

In addition, they did some experiments on the normalized images which we'll talk more about in section 4.4.

UT multiview gaze dataset

UT multiview gaze dataset from the University of Tokyo has 64,000 eye images and 1.152 million 3D images. As such, it's the largest gaze estimation dataset out there as per the number of images. It contains 50 subjects and a total of 160 gaze directions, and for each subject and gaze direction there

Figure 4.4: An unnormalized MPIIGaze eye image (left) and a normalized one. [39]

are 8 images taken from different angles. It's available for academic use for free.

As opposed to the MPIIGaze dataset, UT multiview dataset is composed of images taken under controlled conditions. This means the lighting of the images is relatively uniform, as is the quality and type.

Other datasets

Columbia gaze dataset has 5880 high resolution (5184 times 3456 pixels) images of people looking at predefined directions. The images are taken under controllod conditions and the lighting, quality and type of the images is very uniform. In addition, the authors of the dataset provided some results using the dataset. They were mostly concerned about gaze detection, that is, finding out if the person in the picture is looking directly at the camera. The dataset is freely available. [29]

EyeDiap is a dataset of videos. However, in addition to videos requiring more work to process, the dataset is not directly available for students. [2]

SynthesEyes contains 11,382 3D-generated close-up images of eyes. They use a total of 10 different eye models, 5 male and 5 female. The images are very realistic and of high quality. [36]

Previous results

As feature-based gaze estimation methods are the most popular approach, most of the results utilize them too. However, since we're interested in neural networks, we're more interested in any approaches utilizing them. Until recently, most of the neural network approaches relied on using neural networks with feature measurements as the input. However, during the past couple of years there have been some approaches using convolutional neural networks with the raw images as the input.

Zhang et al [39] approached the problem with a convolutional neural netword to estimate the gaze direction. In addition, they trained several other classifiers for comparison and did a cross-dataset evaluation against other datasets using UT multiview dataset as the training data.

They had multiple approaches with regards to the training data usage. One approach was to train the neural network on the UT multiview dataset and try to use the trained network on other datasets. This resulted in 13.9 degree average deviation when the network was tested with data from MPI-IGaze and 10.5 degrees with Eyediap.

On the other hand, they also trained the network with MPIIGaze data, both in a person-dependent and person-independent way. Naturally, the person-dependent way provided more accurate results with a mean error of slightly over 3 degrees (exact number not given), while the person-independent case gave a mean error of 6.3 degrees.

One thing to note is that they used head pose information and showed that it matters, since performance degraded if it was not given to the network.

Wood et al [36] did a similar approach with the SynthesEyes dataset. They trained the same neural network and achieved some similar results, but improved on others. They came to the conclusion that using additional 3D datasets while training helps with the accuracy, even when the test set was composed of real-world images. This shows using 3D images to augment the data is beneficial.

When they trained the network with UT multiview or SynthEyes datasets alone, the results were worse than those obtained when both datasets were used. The results futher improved when UT multiview and SynthEyes datasets were sampled so that the head pose and gaze directions matched those in the MPIIGaze dataset. This resulted in 7.9 degrees mean error.

Chapter 5

Implementation

In this chapter we'll cover the implementation details and the justify the design decisions.

Dataset

As seen in chapter 4, there are several datasets we can use.

Obviously an important criteria is the accessibility of the dataset. This leaves out EyeDiap since it's not available for students.

In addition, we'd prefer to have comparable results, and for us this means a preference for neural network -based approaches. Zhang et al [39] had an approach based on neural networks, as did Wood et al [36]. The former relied mostly on images taken in realistic environments. In addition, the former presented some person-dependent results which the latter didn't and chose to concentrate on gaze estimation from images under realistic lighting, which is of particular interest to us.

Due to these reasons we chose to follow the experiments by Zhang et al [39].

Data preprocessing

While the dataset was generally of good quality and the authors had already provided both the unnormalized and normalized images, it had some minor issues that needed to be fixed.

The main issue we discovered concerned the number of normalized and unnormalized images. The images were grouped by person and the day they were taken. In two instances the number unnormalized images differed from

the number of normalized images. We notified the authors of the dataset and the issue was fixed.

We were also concerned that the left and right side of the eyes were swapped, but as the direction of the gaze is roughly the same for both the left and right eye, it turned out not to matter. Swapping the left and the right eye provided nearly identical results, as can be seen in figure 5.1.

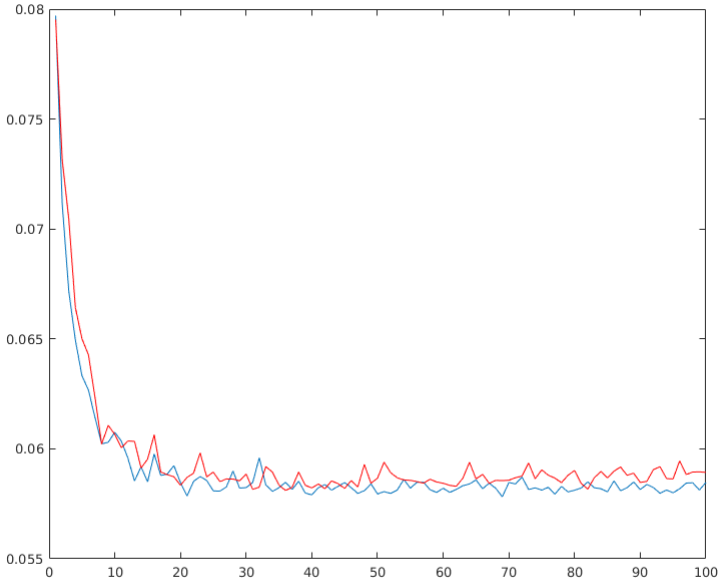

Figure 5.1: Blue line is the training error (radians) when the gaze direction vectors are swapped, red is the original.

Finally, since the number of images per participant varied, we followed Zhang et al [39] by limiting the number of images per person to 3000 to make the dataset more balanced.

Splitting the data

We're mainly interested in person-independent gaze estimation. If we've already decided the network structure to use and have no hyperparameters to tune, we can forego the usage of validation set, and only create training and test sets. This was done by Zhang et al [39].

However, in our situation we want to also create a validation set to estimate the quality of the network we've trained, before selecting the best network and applying it to the test set. Since the number of people in the

selected dataset is quite small (15 people) creating a train/validation/test split becomes more difficult. The low number of people forces us to have relatively few people in the validation set, meaning that the validation error will have a high variance, potentially leading to the selection of a non-optimal configuration.

In addition, the low number of people makes it necessary to do leave-one-person-out testing, since we can't establish a test set large enough to properly estimate the generalized error with 15 people. We decided to divide the data into a training set of 9 people, validation set of 4 people and a test set of 1 person, and to do this for each person separately, so that each person would appear in the test set once. In practise we then, for each hyperparameter configuration, train the network with 9 people and estimate its error rate with 4 people. Then, once we determine the network structure with the lowest error rate, we estimate its actual error with 1 person.

The above procedure requires us to run the hyperparameter optimization 15 times and requires a lot of computational power, but makes sure the variance of the result is minimized.

Another approach would be to first use the person-dependent scenario to optimize the hyperparameters, and then apply those hyperparameters in the person-independent scenario. In this case the person-dependent scenario could be thought of as a cheaper proxy function.

This approach has two main downsides. First of all, it's not clear if the hyperparameters optimized for the person-dependent scenario would work well in the person-independent one.

Second, there would be a problem with the separation of training and test data. In the person-dependent scenario we would be using a data set which contains all the 15 people mixed together and optimize the neural network hyperparameters based on that. After that, we'd use those hyperparameters in the person-independent case, doing leave-one-person-out testing. However, no matter who we choose as the test person, we have, in a sense, already seen them when doing the hyperparameter optimization in the person-dependent scenario. This means we've chosen the hyperparameters, in part, based on the test set. On the other hand, it could be argued that the effect is weak, and the significance of one person to the optimal hyperparameter set is likely to be small.

Initial settings

This section describes the initial settings for our experiments.

User-defined parameters

Perhaps ironically, there are parameters which define how we can optimize the hyperparameters of neural networks, and most of these can't be set programmatically, or at least doing so is very hard.

An important hyperparameter of this type is the computation time, or the number of epochs to fit each neural network. One way to try to bypass this hyperparameter is to use early stopping. However, we still need some maximum number of epochs we can run, in case the hyperparameter optimizer chooses some parameters which make the network learn very slowly, for example a very small initial learning rate.

In addition, we need to decide which hyperparameters to optimize over. Each new hyperparameter causes the number of possible configurations to increase exponentially. The number of hyperparameters to optimize over needs to correspond to the available resources: if we can only afford to do a few iterations, we can only afford to choose very few hyperparameters. Otherwise, too much of the configuration space will be left unexplored and the hyperparameter optimization will not be able to do its intended job.

When doing Bayesian hyperparameter optimization, one more thing we need to decide beforehand is the library to use. Choosing the optimal library would require some knowledge of their characteristics and how well they fit the task at hand.

Network structure

We choose the network used by Zhang et al [39] as the starting point for the network structure. The network has two convolutional layers, both with 5x5 kernels, followed by a fully connected layer. The first convolutional layer has 20 feature maps and the second has 50. Each convolutional layer is followed by 2x2 maximum pooling. The following fully connected layer is composed of 500 nodes. After that, we have the output layer which has two nodes, one for the gaze angle along the x-axis and one for y-axis. All the activation functions are rectifiers, except for the regression layer which has a linear activation function. The head pose information is concatenated to the input vector of the regression layer, meaning that the output nodes actually receive an input vector of length 502 instead of 500.

Hyperparameters to optimize

For the hyperparameters to optimize over, following Snoes et al [33], we considered the following hyperparameters: initial learning rate, momentum,

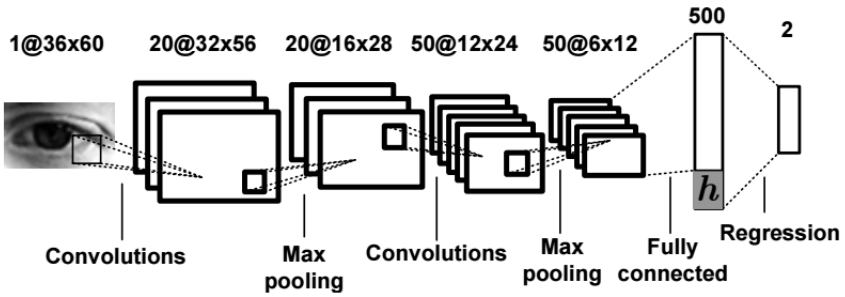

Figure 5.2: Neural network used by [39]

layer sizes, dropout rates and L^2 normalization penalties.

Initial learning rate has almost always been shown to have a significant effect on the result, which is why we include it in the optimization [8].

As to L^2 regularization, as discussed previously, it can be dropped in favor of early stopping. In addition to having the same effect as L^2 regularization, early stopping decreases the total computation time since we won't necessarily train the maximum number of epochs. In fact, during training we noticed that between half and three quarters of training runs ended before the maximum number of epochs. This was usually due to a too large initial learning rate.

We chose 30 epochs as the upper limit for the number of epochs to run. This is because when running the network with the original configuration by Zhang et al [39] the results didn't improve after 30 iterations, as can be seen in figure 5.1. In addition, the lower number of epochs is needed to keep the total computation time reasonable.

For the optimizer, we chose Adam, which is a fairly new optimizer with demonstrated good results when training neural networks. We also drop momentum from consideration, as Adam doesn't have a parameter which would directly correspond to it. [20]

This leaves us with the initial learning rate, layer sizes and dropout rates to optimize over.

All popular hyperparameter optimizers require us to predefine limits to the values being optimized. The maximum and minimum values of the chosen hyperparameters are shown in table 5.1. These values are based on the initial values by Zhang et al [39] and the insight that larger values typically don't hurt the generalization performance as much as smaller values, making them safer to use [8]. As we'll later see, most of the found optimal values fall within these ranges.

hyperparameter name	minimum value	maximum value
learning rate	0.000001	0.01
dropout rate	0	0.9
size of the 1st convolutional layer	20	60
size of the 2nd convolutional layer	40	100
size of the fully connected layer	100	600

Table 5.1: Hyperparameters to optimize over

Some hyperparameters might have an major effect on the results but we intentionally leave them out due to concerns over computation time. These parameters include the kernel sizes of the convolutional layers, regularization methods outside of dropout, L^2 and early stopping and sizes of the pools or strides. In addition, we leave out the number and type of the layers, and predefine the network to have two convolutional layers and one fully connected layer before the regression layer, meaning we won't have any conditional hyperparameters.

Library selection and scripts

Spearmint [30] is perhaps the most used library for Bayesian hyperparameter optimization. However, GPyOpt [4] seems to have roughly the same functionality and adds some of its own, allowing, for example, parallel Bayesian hyperparameter optimization. In addition, GPyOpt has a simpler setup, not requiring MongoDB.

Fabolas [22] seems to have good results, but installing it proved to be difficult. This is probably due to the library being under active development.

SMAC, on the other hand, could be considered a slightly non-Bayesian approach, which would partially defeat our goal to focus on Bayesian methods.

Since our experimental setup requires a lot of computational resources, using parallel optimization is an extremely useful feature for us. Due to these reasons we chose to use GPyOpt.

As for the neural network library, we selected Keras. Keras is a python-based neural network library and is one of the most popular deep learning libraries available [3]. Despite being written in python, it delegates the heavy computations to subroutines written in more efficient languages. It implements all the functionality needed for our experiments. [12]

To utilize parallel optimization, we created some helper scripts. These scripts handle the actual parallelization by querying machine loads, checking that no one is logged into the machines from classroom and by restarting

failed jobs.

Hardware

The hardware we used were Aalto university's computer lab ("maari") computers. They contain Intel Xeon CPU E3-1230 V2 processors, which have 8 cores running at 3.30GHz. In addition, they have 16 GB of RAM and, significantly for us, NVIDIA Quadro K2000 graphics processing units.

In addition, the university offers a computing cluster with more powerful GPUs. However, to avoid queues and to reduce the load on the cluster, I opted to use the computers at the computer lab. The scripts we used made sure no other user was using the computers at the same time to avoid distracting classroom usage.

Reproducing earlier experiments

To reproduce the experiments by Zhang et al [39], we replicated their neural network with Keras and ran it for the given number of iterations (100). We manager to reproduce the results outlined in section 4.4, meaning that training and evaluating the network in a leave-one-person-out manner led to an average error of roughly 6.3 degrees. However, this was after some tuning of the initial learning rate, which was needed since the initial learning rate was not reported by Zhang et al [39]. Using the Keras default of 0.001 as the initial learning rate led to a slightly worse average deviation of 6.6 degrees. The person-specific errors can be seen in figure 5.3.

To measure the person-dependent results, we divided the dataset into a training set (75 percent of the data) and test set, and ran the above descibed neural network. The person-dependent accuracy varied from 3.15 degrees to 3.75 degrees. The error of 3.15 degrees roughly corresponds to the one presented by Zhang et al [39], that is, it's roughly equivalent to the first column titled "MPIIGaze (person-specific)" in figure 5.4. The exact error rate was not reported by Zhang et al [39], but the error appears to be slightly above 3 degrees, which is consistent with the minimum error of 3.15 degrees in our reproduced network.

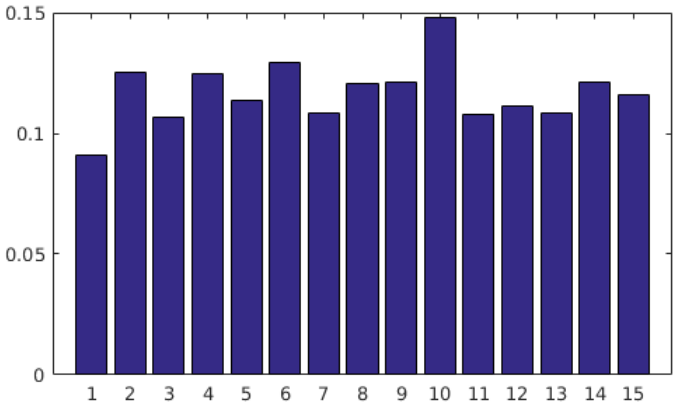

Figure 5.3: Person-specific errors when running the original network for 100 epochs with an initial learning rate of 0.001

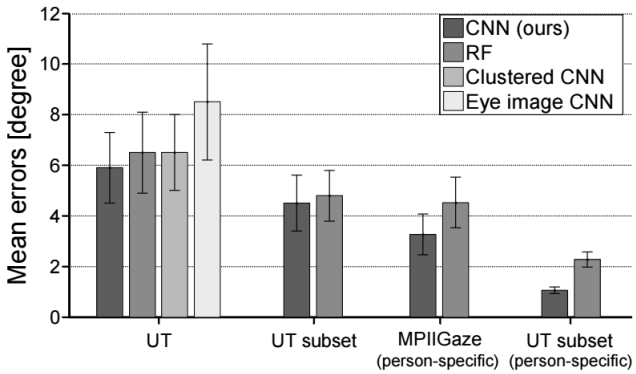

Figure 5.4: Results by Zhang et al. [39]

Chapter 6

Evaluation

In this chapter we'll present the results we achieved and compare them to the previous best.

Person-dependent

To run the hyperparameter optimization, we divided the dataset into train, test and validation sets. Training set consisted of 50 percent of the data, validation and test set of 25 percent each. The validation set was used to select the most promising hyperparameter configuration. After selecting the best configuration it was used to train the neural network using both the previous training and validation data as the new training data (75 percent of the data in total), and the test set was used to get the final error.

After running 20 parallel iterations of hyperparameter optimization with 10 batches each, a total of 200 evaluations, we ended up with a validation error of 3.15 degrees. While this matches the one by Zhang et al [39], it was achieved with only two thirds of the training data, assuming they used a similar split. After training the same network using combined training and validation data for training, the final test error was 2.9 degrees. This is an 8 percent improvement over state of the art, assuming the best result by Zhang et al [39] was 3.15 degrees.

Figure 6.1 shows the validation error in radians during the optimization. The highest error rate was usually caused by a learning rate that was too high, making the network unable to learn anything.

In addition to the normalized images we tried using the unnormalized images, as seen in figure 4.4 on the left, to estimate the gaze direction. At 5.1 degrees average deviation the results were significantly worse, so we didn't pursue this approach further. The neural network settings used can be found

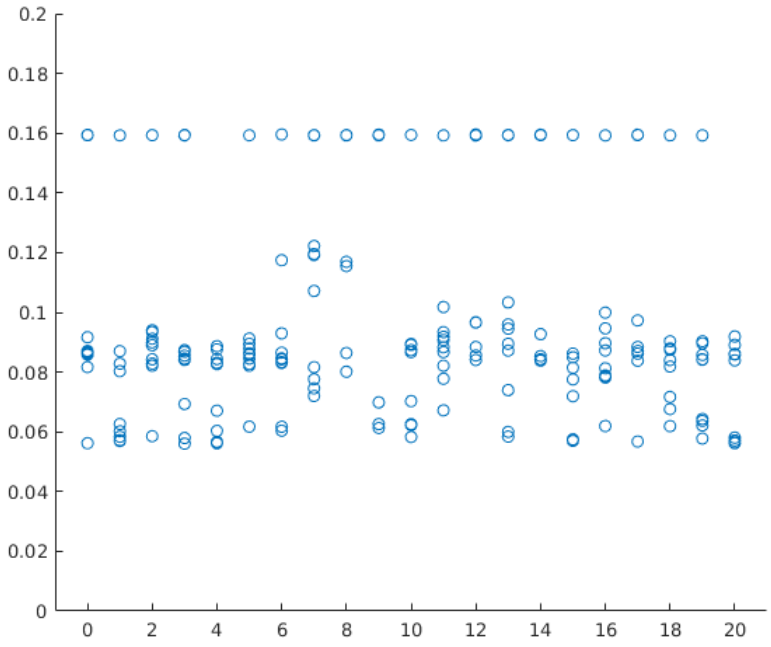

Figure 6.1: Validation set error (radians) when running Bayesian hyperparameter optimization for 20 iterations with a batch size of 10.

in appendix A

The effect of dropout seemed somewhat inconsistent. The best configuration had the dropout rate at nearly zero, while some configurations with nearly the same error rate utilized dropout more. To investigate this, we created an ensemble of neural networks, which should have an effect similar to using dropout. We trained 15 neural network with the same structure as the one used by Zhang et al [39] for 30 epochs and used the average of their predictions. This gave a test error of slightly below 2.8 degrees, which is even better than the one achieved with hyperparameter optimization and is approximately an 11 percent improvement over state of the art.

Person-independent

We ran the person-independent optimization with the validation scheme described in section 5.1.2. For each person, we ran 10 iterations with a batch size of 10. This means we train 100 networks for each person and 1500 in total.

Using Bayesian hyperparameter optimization we achieved an average error of 6.1 degrees. This is a 7 percent improvement over the unoptimized original network and a 3 percent improvement over the results by Zhang et al [39].

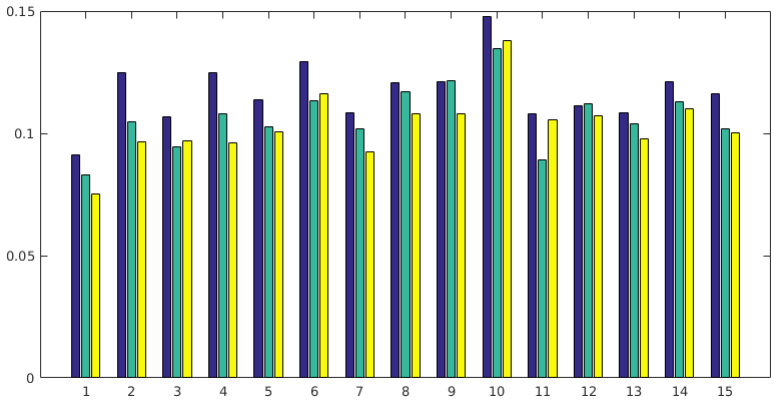

Figure 6.2: Mean error per person: original network (blue), hyperparameter optimized network (green) and neural network ensemble (yellow). See appendix C for the detailed error rates.

As in the person-dependent case, we also applied the idea of an ensemble. For each person we trained 15 networks for 30 epochs and used the mean of their predictions. The structure of the networks was the same as those used by Zhang et al [39]. This resulted in an average error of 5.9 degrees, an improvement of over 6 percent over state of the art.

We also applied the network found to be optimal for the person-dependent scenario to the person-independent scenario. However, with a mean error of 6.3 degrees, there was no improvement compared to the results by Zhang et al [39].

Chapter 7

Discussion

In this chapter we'll outline the most important findings.

Bayesian optimization of neural network hyperparameters

As can be seen from figure 6.1, nearly optimal configurations were discovered fairly early and there were only slight improvements after that. This is consistent with the findings by Klein et at [22] who showed that Bayesian optimization converges to the optimum quickly after escaping any initial plateau. This tendency was possibly further magnified by the usage of parallelization.

Hyperparameter optimization requires a lot of computation power. As such, approaches such as using only part of the training data initially are attractive to reduce the computational load. However, these methods are still not fully productionized and suffer from the downsides associated with using software under constant development.

In addition, hyperparameter optimization has its own parameters which require some understanding and expertise to be tuned. However, doing the hyperparameter optimization in an automated manner does reduce the need for human effort. In addition, less understanding of the hyperparameters and their optimal values are required since the user relies on the optimizer to converge on the optimal values.

Finally, on a more subjective note, I was positively surprised by the ease of usage of some of the hyperparameter optimization libraries.

The optimal configurations found for each person in the person-independent scenario can be found in appendix B. As can be seen, many of the optimal values are similar to each other, although some exceptions exist. For example, the size of the last layer is usually in the high 200's or low 300's. This is

not surprising, since the training data for each run was mostly the same.

Even though many of the optimal configurations were similar to each other, there were exceptions. This suggests we might've benefitted from increasing the number of function evaluations or, alternatively, reducing the number of hyperparameters to optimize over.

Effect of dropout

One interesting finding was the effect of dropout. Namely, many of the best configuration had a dropout rate close to zero. This is in contrast to the results we got by using neural network ensembles, where averaging the predictions of several neural networks achieved a significant improvement in results.

Possible explanation for this might be that the size of the network was too small, and didn't allow for dropout to work well enough. Alternatively, it's possible that applying dropout only to the single fully connected layer constrained its effect. A third explanation might be that dropout causes variance to the results which confuses the hyperparameter optimizer.

Gaze estimation

The results, while state of the art, are still far from what's potentially achievable. As we talked in chapter 4, some approaches can get accuracies of under one degree, making error rates of 5.9 and 6.1 degrees relatively high. However, these results are state of the art when the training data is based on off-the-shelf cameras and natural lighting.

Future improvements

Hyperparameter optimization in the person-dependent scenario improved state of the art results by 7 percent and by 3 percent in the person-independent case. This, perhaps, suggests that the variance between the validation and test data in the person-independent case played a role. It's also possible that the comparison point of 3.15 degrees in the person-independent case was too high, as we had to approximate it from a bar graph.

Some of the above problems could be fixed by increasing the diversity of the dataset by including more people.

In addition, Woods et al [36] showed that combining several gaze estimation datasets can improve results. However, they didn't apply their methods

to the person-independent scenario we have described previously. This suggests that using other datasets to aid the training might be a good way to improve results.

Chapter 8

Conclusions

In this thesis we investigated the effect of Bayesian hyperparameter optimization on gaze estimation neural networks. In addition, we utilized neural network ensembles.

We showed that Bayesian hyperparameter optimization is fairly easy to use but requires a lot of computational power and time. Bayesian hyperparameter optimization can improve results, and we saw a 3 percent improvement on state of the art results.

We saw that hyperparameter optimization requires setting some of its own parameters to work well, which might affect the quality of the results.

Some parameters seemed hard to optimize. The usage of dropout in the optimized networks seemed inconsistent and didn't seem to use all of its potential. This was shown by using a neural network ensemble which saw a 6 percent improvement over state of the art.

For training data, we showed the effect of normalizing it by using the unnormalized training data, which gave a significantly worse result. Also, we showed the importance of having training data with variety. Due to the low number of people in the data set we used, we ended up using methods with a very high computational cost.

We showed that while it's possible to use camera-based gaze estimation, using it for commercial purposes still produces relatively high training errors compared to more invasive methods.

Bibliography

[1] eye structure. http://www.freepik.com/free-vector/eye-anatomy-vector_760161.htm. Accessed: 2016-10-29.

[2] Eyediap overview. https://www.idiap.ch/dataset/eyediap. Accessed: 2016-10-02.

[3] most starred deep learning libraries. https://github.com/aymericdamien/TopDeepLearning. Accessed: 2016-10-06.

[4] AUTHORS, T. G. GPyOpt: A bayesian optimization framework in python. http://github.com/SheffieldML/GPyOpt, 2016.

[5] Baluja, S., and Pomerleau, D. Non-intrusive gaze tracking using artificial neural networks.

[6] Benenson, R. Classification dataset results. http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html, 2016. [Online; accessed 6-September-2016].

[7] Bengio, I. G. Y., and Courville, A. Deep learning. Book in preparation for MIT Press, 2016.

[8] Bengio, Y. Practical recommendations for gradient-based training of deep architectures. *CoRR abs/1206.5533* (2012).

[9] Bergstra, J., and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research 13* (Feb. 2012), 281–305.

[10] Bergstra, J., and Bengio, Y. Random search for hyper-parameter optimization. *J. Mach. Learn. Res. 13* (Feb. 2012), 281–305.

[11] Brochu, E., Cora, V. M., and de Freitas, N. A tutorial on bayesian optimization of expensive cost functions, with application to

active user modeling and hierarchical reinforcement learning. *CoRR abs/1012.2599* (2010).

[12] Chollet, F. keras. `https://github.com/fchollet/keras`, 2015.

[13] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. The loss surface of multilayer networks. *CoRR abs/1412.0233* (2014).

[14] Damelin, S. B., and Miller, W. *The Mathematics of Signal Processing*. Cambridge University Press, January 2012.

[15] Feng Lu, Takahiro Okabe, Y. S., and Sato, Y. A head pose-free approach for appearance-based gaze estimation. In *Proc. BMVC* (2011), pp. 126.1–126.11. http://dx.doi.org/10.5244/C.25.126.

[16] GonzÃ¡lez, J., Dai, Z., Hennig, P., and Lawrence, N. D. Batch bayesian optimization via local penalization.

[17] Hansen, D., and Ji, Q. In the eye of the beholder: A survey of models for eyes and gaze. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2010).

[18] Hutter, F., Hoos, H., and Leyton-Brown, K. An efficient approach for assessing hyperparameter importance. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014* (2014), pp. 754–762.

[19] Jacob, R. J. Eye tracking in advanced interface design.

[20] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *CoRR abs/1412.6980* (2014).

[21] Klein, A., Bartelsn, S., Falknern, S., Hennign, P., and Hutter, F. Towards efficient bayesian optimization for big data.

[22] Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. Fast bayesian optimization of machine learning hyperparameters on large datasets. *CoRR abs/1605.07079* (2016).

[23] Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR abs/1603.06560* (2016).

[24] Mazur, M. A step by step backpropagation example, 2015. Accessed: 2016-08-19.

[25] Morimoto, C. M., and Mimica, M. R. M. Eye gaze tracking techniques for interactive applications.

[26] Nielsen, M. A. Neural networks and deep learning. Book in preparation for Determination Press, 2015.

[27] Rasmussen, C. E., and Williams, C. K. I. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[28] Ruder, S. Overview of gradient descent optimization algorithms. `http://sebastianruder.com/optimizing-gradient-descent/index.html`, 2016. [Online; accessed 7-September-2016].

[29] Smith, B. A., Yin, Q., Feiner, S. K., and Nayar, S. K. Gaze locking: Passive eye contact detection for human?object interaction. In *ACM Symposium on User Interface Software and Technology (UIST)* (October 2013), pp. 271–280.

[30] Snoek, J. Spearmint. `https://github.com/HIPS/Spearmint`, 2016.

[31] Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25 (NIPS 2012)* (2012).

[32] Snoek, J., Swersky, K., Zemel, R. S., and Adams, R. P. Input warping for bayesian optimization of non-stationary functions. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014* (2014), pp. 1674–1682.

[33] Snoes, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M. M. A., Prabhat, and Adams, R. P. Scalable bayesian optimization using deep neural networks, 2015.

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

[35] Vicente, F., Huang, Z., Xiong, X., la Torre, F. D., Zhang, W., and Levi, D. Driver gaze tracking and eyes off the road detection system. *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS* (August 2014).

[36] Wood, E., Baltrusaitis, T., Zhang, X., Sugano, Y., Robinson, P., and Bulling, A. Rendering of eyes for eye-shape registration and gaze estimation. In *Proc. of the IEEE International Conference on Computer Vision (ICCV 2015)* (2015).

[37] Xu, B., Wang, N., Chen, T., and Li, M. Empirical evaluation of rectified activations in convolutional network. *CoRR abs/1505.00853* (2015).

[38] Young, L. R., and Sheena, D. Methods and designs: Survey of eye movement recording methods. *Behavior Research Methods and Instrumentation* (1975), 397–429.

[39] Zhang, X., Sugano, Y., Fritz, M., and Bulling, A. Appearance-based gaze estimation in the wild. In *Proc. of the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2015), pp. 4511–4520.

Appendix A

Neural network configuration for unnormalized images

The networks structure used for the original unnormalized images consists of three convolutional layers. The convolutional layers had a stride of two. The first convolutional layer had 20 feature maps and a kernel of size 7x7, the second had 50 feature maps and a 5x5 kernel, the third had 30 feature maps and a kernel of 5x5. The convolutional layers were followed by a fully connected layer with 100 nodes, and finally the regression layer.

The optimizer used was Adam with default settings, which can be found in the Keras documentation. The activation function was the rectifier, apart from the last (regression) layer which had a linear activation function. Batch size was 32.

Appendix B

Neural network configurations used for each person

learning rate	0.000300572005194
dropout rate	0.0408368221974
size of the 1st conv. layer	58
size of the 2nd conv. layer	71
size of the fully connected layer	319

Table B.1: Person 1 optimal settings

learning rate	0.0000840406932124
dropout rate	0.330695861711
size of the 1st conv. layer	21
size of the 2nd conv. layer	90
size of the fully connected layer	247

Table B.2: Person 2 optimal settings

learning rate	0.000539386797135
dropout rate	0.426473492738
size of the 1st conv. layer	24
size of the 2nd conv. layer	67
size of the fully connected layer	269

Table B.3: Person 3 optimal settings

learning rate	0.000515620565805
dropout rate	0.233327202217
size of the 1st conv. layer	36
size of the 2nd conv. layer	62
size of the fully connected layer	239

Table B.4: Person 4 optimal settings

learning rate	0.0000659439500235
dropout rate	0.36024738319
size of the 1st conv. layer	27
size of the 2nd conv. layer	79
size of the fully connected layer	227

Table B.5: Person 5 optimal settings

learning rate	0.0000415222298
dropout rate	0.818547634697
size of the 1st conv. layer	26
size of the 2nd conv. layer	100
size of the fully connected layer	374

Table B.6: Person 6 optimal settings

learning rate	0.00003950225
dropout rate	0.531012951694
size of the 1st conv. layer	49
size of the 2nd conv. layer	67
size of the fully connected layer	308

Table B.7: Person 7 optimal settings

learning rate	0.000591348345163
dropout rate	0.399982064071
size of the 1st conv. layer	50
size of the 2nd conv. layer	59
size of the fully connected layer	553

Table B.8: Person 8 optimal settings

learning rate	0.00100476755545
dropout rate	0.121857491758
size of the 1st conv. layer	29
size of the 2nd conv. layer	95
size of the fully connected layer	526

Table B.9: Person 9 optimal settings

learning rate	0.000282073478782
dropout rate	0.485726437522
size of the 1st conv. layer	60
size of the 2nd conv. layer	71
size of the fully connected layer	318

Table B.10: Person 10 optimal settings

learning rate	0.0002084978408
dropout rate	0.858055151886
size of the 1st conv. layer	26
size of the 2nd conv. layer	71
size of the fully connected layer	510

Table B.11: Person 11 optimal settings

learning rate	0.000134272491519
dropout rate	0.00395279305135
size of the 1st conv. layer	56
size of the 2nd conv. layer	92
size of the fully connected layer	279

Table B.12: Person 12 optimal settings

learning rate	0.000146735049892
dropout rate	0.00203331016666
size of the 1st conv. layer	32
size of the 2nd conv. layer	77
size of the fully connected layer	310

Table B.13: Person 13 optimal settings

learning rate	0.0000962904984212
dropout rate	0.736861642078
size of the 1st conv. layer	26
size of the 2nd conv. layer	42
size of the fully connected layer	466

Table B.14: Person 14 optimal settings

learning rate	0.000262904372843
dropout rate	0.0171213567484
size of the 1st conv. layer	52
size of the 2nd conv. layer	97
size of the fully connected layer	239

Table B.15: Person 15 optimal settings

Appendix C

Person independent error rates

person	error rate
1	0.0752
2	0.0966
3	0.0971
4	0.0961
5	0.1005
6	0.1161
7	0.0926
8	0.1079
9	0.1078
10	0.1377
11	0.1054
12	0.1070
13	0.0976
14	0.1098
15	0.1001

Table C.1: Neural network ensemble

person	error rate
1	0.0830
2	0.1045
3	0.0945
4	0.1078
5	0.1025
6	0.1131
7	0.1018
8	0.1171
9	0.1217
10	0.1346
11	0.0890
12	0.1120
13	0.1038
14	0.1128
15	0.1017

Table C.2: Hyperparameter optimized networks

person	error rate
1	0.0910
2	0.1250
3	0.1066
4	0.1246
5	0.1136
6	0.1294
7	0.1086
8	0.1209
9	0.1213
10	0.1479
11	0.1080
12	0.1114
13	0.1085
14	0.1210
15	0.1160

Table C.3: Original network by Zhang et al with a learning rate of 0.001, average over 5 runs