Eötvös Loránd University
Faculty of Informatics
Department of Software Technology and Methodology

Hyperparameter optimization of Machine Learning algorithms

Zoltán Tősér
PhD Candidate, MSc
dr. habil. András Lőrincz
Senior Research Fellow, PhD

Tamás Nyíri
Computer Science BSc

Budapest, 2016

Contents

1 Introduction

Machine Learning can be defined as a field of study that gives computers the ability to learn without being explicitly programmed[1]. It's a field rising in popularity, particularly with one of its subfields, "Artificial Neural Networks" having solved a wide variety of problems that long seemed intractable with conventional methods.

In this thesis we are going to look at the example of gaze estimation, and use this example to learn about the Data Science pipeline, starting with the task of data augmentation and finishing with a presentation of the final results.

We will introduce the reader to some of the main ideas involved in machine learning and demonstrate the learning procedure with the help of two popular Machine Learning algorithms: the Support Vector Machine (SVM) - a simple, yet powerful "off-the-shelf" supervised learning algorithm and a Deep Convolutional Neural Network (DCNN) - a biologically-inspired model, tailor-made for computer vision problems.

For almost every machine learning algorithm there are open questions regarding the tuning of their parameters for optimal performance. Our way of trying to solve this problem will be Bayesian Optimization - an automated method for hyperparameter optimization. We will showcase this method on the free parameters of the SVM algorithm.

A large part of the code uses various third-party softwares and libraries. This is necessary given the complex nature of our problem. As this software is intended for use by professionals/researchers from the field of computer science, creating a graphical user interface (GUI) was not a priority. The software is primarily a library, its usage is demonstrated with an example script. This script has a command line interface (CLI).

There were multiple objectives in mind when writing this thesis. The software coupled with this text can serve as an introduction to the field of Data Science. It can also give insights to the pros and cons of using DCNN's as opposed to SVM's, or using automated versus manual hyperparameter search techniques on certain types of learning problems.It was also used for running numerical simulations for a research paper, which was submitted to the KI 2016 German Conference on Artificial Intelligence [2]. It provided results for multiple novel techniques proposed by the Neural Information Processing Group [3].

2 User's Guide

2.1 Motivation

A wide variety of disciplines can benefit from the tracking of the human gaze, ranging from medicine to marketing research. Gaze tracking can provide us with information about user preferences, it can help us interpret social interactions, and we can use that information to gain more understanding about or to better interact with the user.

2.2 Important Concepts

In this section, we are going to briefly, and informally introduce the user to some of the more important topics in the field of machine learning with emphasis on the techniques and methods used in our software.

Special thanks to MIT, Stanford, and Coursera for providing education free of charge. They helped a great deal in absorbing the material used throughout this thesis.[4][5][6][7]

2.2.1 Machine Learning

There exist some problems in the world that we know can be solved, but have no idea how to write an explicit algorithm to solve them. Examples include facial recognition, speech recognition and autonomous driving, just to name a few. People tried to solve these problems with traditional algorithmic approaches, but while those explicit methods may work great on "simple" problems like the sorting of a list, they reliably fail at the "harder problems", not to mention that they are overly complicated and hard to develop.

Luckily, explicit algorithms are not the only way to go. Evolution did not teach man to play chess. Evolution gave man the ability to learn chess by trial and error. That's an implicit approach. It's a statistical approach, using examples and the outcomes of those examples to teach the behavior which is most likely to lead to success.

2.2.2 Supervised vs Unsupervised learning

Machine Learning can be divided into three categories: Supervised Learning, Unsupervised Learning and Reinforcement Learning.

In Supervised Learning, we are supplying our model with both the kind of data we are trying to make predictions about and the output we expect them to produce when faced with that data. We call these expected outcomes "labels".

Reinforcement Learning basically applies Behaviorist Psychology techniques to machine learning problems. It consists of Agents, Environment, Actions and Rewards. Agents try to take Actions in relation to their Environments in a way that maximizes the cumulative Reward. The big difference between Reinforcement Learning and Supervised Learning is the amount of control we exercise over our model. In Supervised Learning, as the name suggests, we constantly try to supervise the model and nudge it in the right decision. In Reinforcement Learning, we let it learn from its own mistakes.

In Unsupervised Learning, we don't have the luxury of having labels nor any other kind of reinforcement. These kind of techniques work by exploiting the structure inherent in the data itself. An example of Unsupervised Learning would be Clustering, where the algotiyhm divides the data into groups according to the similarity of the features they contain.

In this thesis, we will use Supervised Learning.

2.2.3 Classification vs Regression

We can further divide Supervised Learning into two groups according to the kind of output they produce. If their output can only take on finite discrete values, we call them Classification algorithms, otherwise we call them Regression algorithms.

Gaze estimation is an example of a Regression problem.

2.2.4 The problem of underfitting and overfitting

In every sample we take from real world data, there will be certain amount of information pertaining to the problem we want to solve but also some irrelevant information, maybe even false information that could lead our model astray.

When we fail to utilize useful information that could help our model make better predictions, it's called "Underfitting" (high variance).

Conversely, when we use too much data, even the irrelevant information or the random noise present in our training data, we call that "Overfitting" (high bias). A good strategy for avoiding Overfitting would be to try and minimize the amount of unnecessary information involved in the training data with methods like Principal Component Analysis (PCA). An example of PCA would be converting a color image to grayscale. It is also one of the methods we will use to reduce the variance of our model.

2.2.5 Data Augmentation

One thing we can do to supply more information to our model and thus reduce its underfitting without actually collecting more data is to generate new data, aggregated from the ones available by the application of some sort of transformation on the original data. The type of transformation depends on the particular problem we are investigating. If our input consists of real numbers, it could be the creation of a new variable as an algebraic function of some of the original variables or if our inputs are images, it could be the creation of new images by the application of a geometric transformation.

The data augmentation method we will use in this work will consist of yaw and pitchwise rotations of the original images.

2.2.6 Model validation

After we are done with the training of our machine learning algorithm of choice, we need to find out how accurate our model turned out to be. This can be done in several ways, but the basic principle remains. We run our model on some input data and in return, it provides us with predictions. We then compare the output of the model with the expected output. (To find out more about the nature of this comparison, see: "Metrics for quantifying "prediction error"")

We need to avoid the mistake of trying to validate our model with the same data it was trained on, as this could lead to overtraining. A solution to this problem is to seperate our database into a number of non-overlapping folds, then validate on each one of these folds separately after training the model on the rest of the folds. In the machine learning community this is called "Leave one out crossvalidation".

We will use "Leave one out crossvalidation" for the validation of our model or more precisely "Leave one subject out crossvalidation", dividing our database into as many pieces as the number of subjects it contains and validating on each subject separately.

2.2.7 Metrics for quantifying "prediction error"

The two metrics used in this work to quantify the prediction error of our model are "Root-mean-square Error" (RMSE) and "Mean Error". The equations used to compute these errors are the following:

- RMSE: $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$
- Mean Error: $\frac{1}{n}\sum_{i=1}^{n}|(y_i - \hat{y}_i)|$

Where n is the number of examples validated on, y_i is the i^{th} output expected (the "correct" output) and $\hat{y}_i$ is the i^{th} output predicted by our model.

2.2.8 Suppor Vector Machine and Support Vector Regressor

The Support Vector Machine is a simple but powerful machine learning algorithm. It's main idea is the following: find the hyperplane that maximizes the margin of seperation between the data points belonging to different labels. It was originally developed to solve classification problems.

The Support Vector Regressor is a version of the Support Vector Machine used for regression problems. It inherits the main idea of a Support Vector Machine, but modifies it to be able to deal with regression. We won't go into detail about the actual process of teaching a Support Vector Regressor, but we will broadly characterize some of the parameters pertaining to it. If you want to learn more about the implementation of the Support Vector Regressor, please refer to the documentation of LIBSVM[8]

In the Support Vector Regressor, the parameter ε is responsible for setting the maximum distance our model is permitted to deviate from the optimal margin. Its value is negatively correlated with the number of support vectors used in the construction of the margin.

Another important parameter used in both the classifier and regressor version of the machine is the parameter C which is responsible for the regularization of the model. The higher the value of C, the more prone the SVM is to overfit, the lower the value of C, the more prone it is to underfit. It decides how much our input variable is permitted to cross the margin while still remaining on the same side.

The main idea of the Support Vector Machine can only be used on linearly seperable problems. As a solution to this, we could use something called a Kernel which elevates our data points into a higher dimensional space thus making it linearly seperable. Our Support Vector Machine doesn't use any kernel, or a different way of putting it is that it uses a Linear Kernel.

2.2.9 Deep Convolutional Neural Networks

A Neural Network is a special type of learning algorithm, inspired by the biological brain. It consists of layers of artificial neurons connected together to form a network, much like biological neurons do. These artifical neurons are simple computational units that work by taking the sum of their weighted inputs, multiplying each input with its corresponding weight, then adding a final, so-called "Bias Term" to the resulting sum. This is the most basic idea of an artificial neuron and is called a "Linear neuron". It's also fairly useless in practice. A useful addition to this idea would be to apply a transformation to the resulting output before sending it as the input of the next neuron. This transformation is almost always a nonlinear one, and in those cases, we call it a "Nonlinearity".

Putting it all together, the resulting equation of our artificial neuron becomes: $z = f(\sum_{i=1}^{n} w_i * x_i + b)$ where we have n inputs coming into our neuron and n weights to regulate the strength of those inputs. The term x_i stands for our i^{th} input, while w_i stands for the weight associated with that input; b is our Bias term and f is our Nonlinearity. This gives us the final output of the neuron, z which will be either used as an input to other neurons, or as an output of the network. When all these different artificial neurons connect to one another to form our artificial neural network, the resulting structure will always obey some invariant rules. There will be one and only one Input Layer with the same number of neurons (same shape) as our input and one and only one Output Layer with the same number of neurons (same shape) as our output, finally one or more Hidden Layers between the them. The term "Deep Network" is a shorthand for a Neural Network with more than one Hidden Layer. Conversely, neural nets with only one hidden layer are called "Shallow Networks"

The implementation of Artificial Neural Networks follows a pipeline similar to the one described here:

- Define the structure and hyperparameters of your network.
- Transform your input data and its corresponding labels into the shape of your Input and Output Layers respectively.
- Initialize your net with random weights. (Or load the weights of a previously trained net to continue its progress.)
- In a loop:
  - Forward propagate through the network to get a hypothesis.
  - Compute the cost function applied to the hypothesis and the expected output.
  - Backpropagate through the network to compute the gradients of the cost function.[9]
  - Use an optimization method like Gradient Descent, to try and minimize the cost function.

To learn more about the exact implementation of the techniques mentioned above, consult the documentation of Theano[10].

Deep Convolutional Neural Networks are special kinds of Deep Networks, inspired by neuroscientific findings about the visual cortex; the part of the brain responsible for vision. The visual cortex processes information in a hierarchical way. It first detects low-level features like edges, then slowly builds up to more and more complex shapes until it is able to perform very high level tasks like face recognition.

Convolutional Networks include special layers called "Convolutional Layers", which have an assumption about the nature of the input (namely, that they are images) and thus arrange their neurons into three dimensions: width, height, and depth, where width and height stands for the width and height of the input image and depth stands for the color channels of its pixels. They also utilize a variable number of filters, each with variable size that scan different parts of the input image, producing 2D activational maps and learning local features from these maps. The number and sizes of the convolutional filters are hyperparameters of the model.

After each convolutional layer we can add another special layer called "Pooling Layer", which downsamples our output and by doing that, acts as a regularizer. It also contains two new hyperparameters per layer, having to do with the Spatial Extent and the Stride of the transformation it performs.

Another layer that is widely used for regularization is the Dropout Layer. It works by randomly dropping out parts of its architecture at each training case and then sampling from an exponential number of these models produced. The hyperparameter associated with this layer determines the amount of neurons the layer drops out during the generation of these thinned-out models.

The convolutional part of our Deep Network will consist of two Convolutional Layers, each of them followed by a Pooling Layer. After the convolutional part, we will insert a Dropout Layer, a Fully Connected Layer, another Dropout Layer, then a second and third Fully Connected Layer, the last one being our Output Layer, producing two outputs: the predicted angles in the x and y dimensions of the gaze associated with the input picture.

2.2.10 Manual vs Automatic hyperparameter optimization methods

Most of the mainstream machine learning algorithms use multiple variables, so-called hyperparameters that can influence their performance a great deal but are very hard to optimize in advance. There do exist some rules-of-thumb for most of these hyperparameters, but their optimal configuration could be highly dependent on factors like the particularities of the training data.

One way to deal with these hyperparameters is to adjust them manually. Try to determine some reasonable configurations, then run the learning algorithm on your dataset with some combinations of those configurations and see which gives you the best performance. This method is preferred when a reasonable number of iterations with an automatic optimization method would take too long. This is the method we used for the tuning of our Deep Convolutional Neural Network.

The other option is Automatic Optimization. The most basic Automatic Optimization method is Grid Search. To perform Grid Search, we need to choose a few possible values for each of our hyperparameters, then simply try out all the combinations of those values. The obvious problem with this approach is that it's prone to combinatoric explosion.

A slightly better approach would be to define a domain for all the hyparameters and randomly sample these domains for each hyperparameter, then run the algorithm with the sampled values. This is not an exhaustive method, but the upside is that it can be stopped at any time, and is expected to give much better results when we don't have much time.

The method we used for finding the optimal configuration of our Support Vector Machine was Bayesian Optimization. This method also starts with random samples, but the difference is that instead of blindly sampling from the sample space, it has an assumption about its statistical distribution and using that assumption, can learn from its previous trials. It uses a model called Gaussian Process, which contains a weak prior saying (broadly speaking) that similar inputs produce similar outputs. After training with specific hyperparameters, the algorithm will have a high confidence about the efficacy of the model with parameters close to the ones previously sampled, and a high variance for regions far away from any previous samples. Bayesian Optimization can be regarded as a meta machine learning algorithm. The training data it recieves comes in the form of outputs from the machine learning algorithm it tries to optimize.

2.3 Requirements

2.3.1 Hardware requirements

- **Mandatory:**
  - A computer capable of running the software requirements
  - Input and output devices
- **Optional:**
  - At least 8GB RAM (depends on the size of the training database and parameters of the learning algorithm used)
  - A CUDA-capable graphics card

2.3.2 Software requirements

- **OS:** Linux
- **Open-source 3rd party softwares:**
  - Operating System: Ubuntu

  - Python 2.7
  - CMake
  - Pip (Pip Installs Packages)
  - Opencv (libopencv)
  - Scikit-image
  - Scikit-learn
  - Termcolor
  - Hyperopt
  - Pymongo 2.1.1.
  - Numpy
  - Scipy
  - Python Imaging Library (PIL)
  - Matplotlib
  - Theano
  - Lasagne
  - cPickle

- **Proprietary 3rd party softwares:**
  - Optional: ZFace (Matlab libraries for 3D Mesh generation from 2D videos)
  - Optional: Matlab
- **The individual requirements of the above softwares**
- **Optional requirements:**
  - Nvidia CUDA (speeds up our computations by using the graphics card for the training of our regressor)

2.4 How to install

If you don't have all the above requirements already installed on your system, you need to run "pip install -r requirements.txt" which should take care of all the missing python modules. If something is still missing, the problem will most likely be resolved by using "pip" to install the missing python extensions, as they appear on your screen after an unsuccessful attempt at running the software (see: "How to use"). If

it's not a python module you are missing, then you will most likely have to install that requirement manually (for example "sudo apt-get install libopencv-dev" for OpenCV or follow the instruction on the website: https://cmake.org/install/ for CMake).
In order to be able to use the software responsible for the training of the support vector regression, the headpose and eye tracking used in this work, it needs to be built. The software is located in the folder "gaze_svm", which contains a subfolder "src" with a file named "options.cpp" in it, which you will have to open in a file editor and change the line describing location of the databases in it to the path of your project, followed by "/dbs", where the databases for this project are located. After this step is done, the source code can be built by CMake, by opening a terminal and typing "cmake CMakeLists.txt", then "make". If everything goes well, this should produce an executable file, called "half-face-tracker"
Supposing you have a GPU capable of running Nvidia CUDA[11], you will have to tell Theano to use it. To make this configuration, go to your "$HOME" folder and create a file named ".theanorc" with the following text[12]:

```
[global]
floatX = float32
device = gpu0

[lib]
cnmem = 1
```

If everything goes well, whenever you import the Theano module in python, you will see something like this message:

```
Using gpu device 0: GeForce GTX 580
```

2.5 How to use

Open a terminal. Change directory to where your "main.py" script is located. Type in: "python main.py". Follow the instructions on the screen. You will be greeted with a main menu, and will have to type in one of the choices present in it to move forward, or exit the program. This structure remains in the submenus as well. When asked for the name of a database or a logfile, you won't have to type in the whole path leading to it, only the name of the logfile or database folder located inside the folders "logs" and "dbs" respectively. When you run the preprocessing or one of the machine learning algorithms, the algorithm will run in your terminal, printing out its progress and will exit to the main menu as soon after its completion. The parts of the program responsible for plotting the results will display the generated plot on the screen, and will hang the program until you close it. Then it will send you back to the main menu. For further information about the inner workings of the program, consult the "Developer's Guide".

This project has a natural folder structure of the following fashion:

- dbs
    - elte1
        - 0001
        - ...
        - 0021
        - contents
    - elte2
        - 0001
        - ...
        - 0012
        - contents
    - elte3
        - 0001
        - ...
        - 0019
        - contents
    - cave
        - 0001
        - ...
        - 0056
        - contents
    - models
        - ...
- gaze_svm
    - src
    - lib
    - CMakeLists.txt
- logs
- last_net_convergence.log

- ...
- zface
- ...
- data
- ...
- models
- ...
- generate_zface_before_rotation.m
- preprocess.py
- hyperopt_search.py
- dcnn.py
- plot.py
- main.py
- requirements.txt
- results.txt

This structure has to be kept in order to ensure proper functioning of the program. To find out more information about the folders and files listed above and the relationship between them, consult the Developer's Guide.

2.6 Data formats

2.6.1 Dumped files and models

The deep convolutional network will serialize and save the data it processed during its first run, saving the name of the database and the interocular distance associated with the images produced in its name. This way, it won't have to process the database all over again at the beginning of every new training process, it can just load a previous dump of it, and work with that. These dump files have the extension ".p" and are contained in the folder "data".

The network is also capable of saving its weights obtained at the end of a training process, and can later load them as the initial weights at the beginning of a new learning process. The saved weights are contained in the folder "models".

2.6.2 Raw database

The software only works with the particular databases mentioned in this section. These databases are located in the "/dbs/" folder. We will use four distinct databases, three of them made by the NIPG group ("ELTE1", "ELTE2", "ELTE3"), and one of them made by Columbia University ("CAVE"):

- "ELTE1", "ELTE2", "ELTE3":
  These are very similar databases. All three consist of images in ".png" format that we have previously cut out from videos (using VLC media player) in which the subjects looked directly into the camera while rotating their heads as far as they could comfortably in the yaw/pitch dimensions.
  In all three databases, we placed the pictures obtained in the previous step into different subfolders with different names, grouping them by subjects. Most of the subjects try various methods of distorting the view to their eyes, such as squinting and keeping their eyes wide open, in order to help the generalization capabilities of the final model (see: "The problem of underfitting and overfitting").
  An important fact about these pictures is that the absolute gaze of all subjects in all positions is zero in both the x and y dimensions, by design. This was achieved by asking the participants to keep their eyes fixed on the camera lense at all times while rotating their heads.
  The biggest differences between the three databases are the size and definition of their images, the number of subjects contained in them and the variance in environment/lighting conditions present on the pictures they contain. If we wanted to rank the databases by the amount of variance in them, it would be something like: "ELTE1" $>$"ELTE2" $>$"ELTE3". Depending on the particular task we want to use them on, we have to choose the appropriate database for the job. The ELTE databases have been provided by The Neural Information Processing Group of Eötvös University, Budapest.[3] See Figure 1 for examples of the dataset.

- "CAVE":
  These images are all in ".jpg" format, with a resolution of 5184*3456 pixels, much sharper than the ones in the ELTE databases. This means more information, but also more memory usage and longer computational time. For our purposes, we chose to resize them to a more manageable 960x640 pixels.
  The database contains 56 different subjects and different head rotations per subjects. The subjects are located in seperate subfolders by default.
  The production of these images are a bit more complicated. The makers of this database were very conscious of including subjects with a wide variety of

ethnicity, different genders, and people with and without glasses.
Another difference in this database is in the methodology of producing the images. The database consists of pictures taken in a very controlled manner and on which the subjects' headposes and gaze directions take on predefined discrete values. The gaze and headpose angles are contained in the filenames. The CAVE database has been provided by Columbia University, New York.[13] See Figure 2 for examples of the dataset.

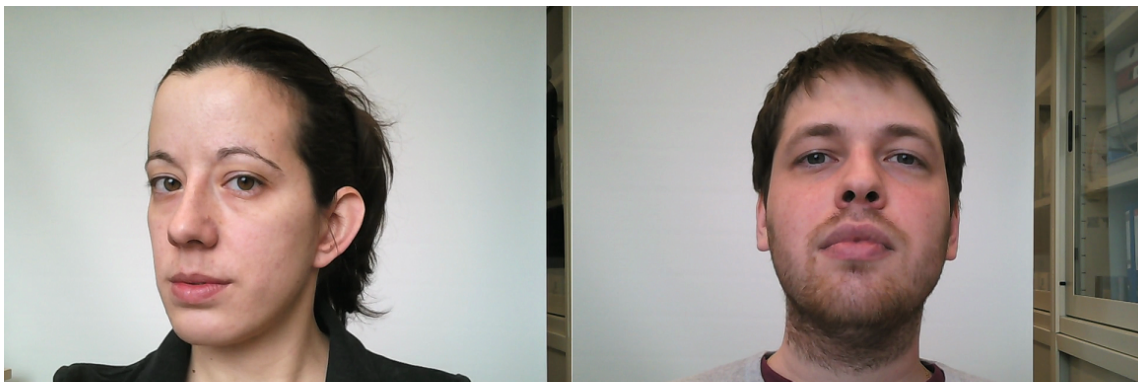

Figure 1: Example images from the "ELTE3" database.

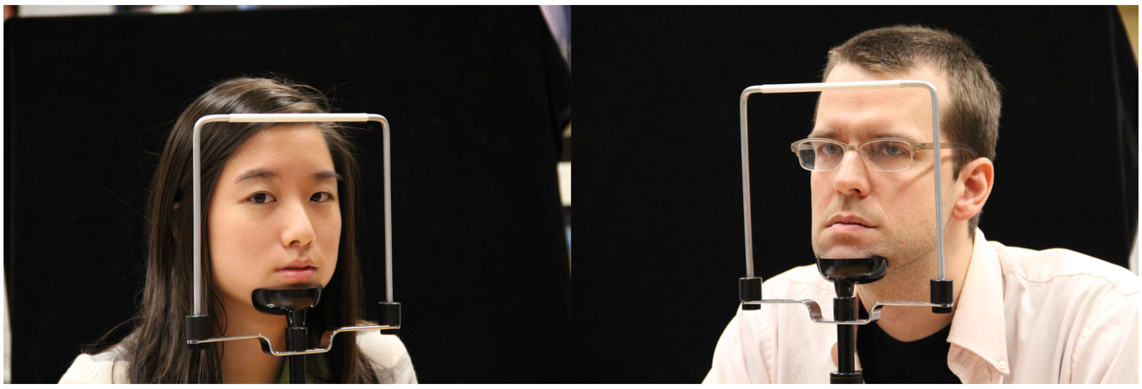

Figure 2: Example images from the "CAVE" database.

2.6.3 Processed database

The augmented databases consist of pictures with a random amount of yaw and pitch-wise rotation. The final angles produced were written in the end of the filename for every image after the string "_processed_". In addition to the pictures, we also provide the facial keypoint markers corresponding to the rotated images, marked by the string "_0.txt" at the end of the filename. This will later be used by both of our machine learning algorithms to help track facial keypoints on the images. As a byproduct of the method used, the process also generates the following files[16]:

- ctrl2D: The 2D locations of the measured facial landmarks.
- mesh3D: The reconstructed dense 3D mesh.
- mesh2D: Projection of the 3D mesh to 2D.
- pars: Shape parameters, in the following order:
  - (1) uniform scaling
  - (2-3) translation x, y direction
  - (4-6) headpose using Euler angles
    pitch(4)-yaw(5)-roll(6) in radians
  - (7-) non-rigid PDM (Point Distribution Model) parameters

2.6.4 Logs

The following files are located in the “logs” folder, generated by the program when running the machine learning algorithms. One of them is used for the plotting of the SVR’s hyperoptimization, and is named by the user before the start of the optimization algorithm, the other one is used for the plotting the convergence of the last neural network trained and is automatically named “last_net_convergence.log”.

- hyperoptimization log:
  - iod: Interocular distance for the resizing of the images.
  - feature_id: 1 for HOG (Histogram of Oriented Gradients), 2 for LBP (Local Binary Patterns).
  - feature_param: Window size for the HOG/LBP.
  - num_imgs_per_train_subj: Number of images to train with.
  - num_imgs_per_test_subj: Number of images to test with.
  - num_imgs_per_pers_subj: Number of images to personalize with.
  - num_rots_per_train_imgs: Number of rotations per training images.
  - num_rots_per_test_imgs: Number of rotations per test images.
  - num_rots_per_pers_imgs: Number of rotations per personalization images.
  - svr_eps: Epsilon parameter of the Support Vector Regressor.
  - svr_c: C parameter of the Support Vector Regressor.
  - svr_p: P parameter of the Support Vector Regressor.
  - db_type: Type of database to use.

  - rmse/mean_error: Root-Mean-Squared Error/Mean Error.
  - use_headpose: Use headpose information or not.
  - use_eyes: Use eye markers or not.

- "last_net_convergence.log":
  - train_rmses: A list of the training rmse's produced during the last validation of the last training.
  - valid_rmses: A list of the validation rmse's produced during the last validation of the last training.

The following files are located in the project folder, in a file called "results.txt" generated by the program when running the machine learning algorithms. They store detailed information about both the SVR's and the DCNN's previous trainings.

- SVR:
  - iod: Interocular distance for the resizing of the images.
  - feature_id: 1 for HOG (Histogram of Oriented Gradients), 2 for LBP (Local Binary Patterns).
  - feature_param: Window size for the HOG/LBP.
  - num_imgs_per_train_subj: Number of images to train with.
  - num_imgs_per_test_subj: Number of images to test with.
  - num_imgs_per_pers_subj: Number of images to personalize with.
  - num_rots_per_train_imgs: Number of rotations per training images.
  - num_rots_per_test_imgs: Number of rotations per test images.
  - num_rots_per_pers_imgs: Number of rotations per personalization images.
  - eps: Epsilon parameter of the Support Vector Regressor.
  - c: C parameter of the Support Vector Regressor.
  - p: P parameter of the Support Vector Regressor.
  - ground truth x: Value of the expected value for the gaze in the x dimesion.
  - estimated x: Value of the estimate value for the gaze in the y dimesion.
  - ground truth y: Value of the expected value for the gaze in the x dimesion.
  - estimated y: Value of the estimated value for the gaze in the y dimesion.
  - Gaze X RMSE: RMSE of the gaze estimation in the x dimension.
  - Gaze Y RMSE: RMSE of the gaze estimation in the y dimension.

  - Gaze RMSE: RMSE of the gaze estimation.
  - Gaze X mean error: Mean error of the gaze estimation in the x dimension.
  - Gaze Y mean error: Mean error of the gaze estimation in the y dimension.
  - Gaze mean error: Mean error of the gaze estimation.

- DCNN:
  - desired_iod: Interocular distance for the resizing of the images.
  - db_type: Type of database used.
  - lambda_l2: Lambda parameter of l2 regularization.
  - dropout1: Dropout rate for first dropout layer.
  - dropout2: Dropout rate for second dropout layer.
  - h1_neurons: Number of neurons in first hidden layer.
  - h2_neurons: Number of neurons in second hidden layer.
  - es_patience: Patience of EarlyStopping method.
  - batch_size: Batch size of gradient descent.
  - train_imgs: Number of images used for training.
  - test_imgs: Number of images used for testing.
  - pers_imgs: Number of images used for personalization.
  - train_rots: Number of rotations per training images.
  - test_rots: Number of rotations per test images.
  - pers_rots: Number of rotations per personalization images.
  - use_eyes: Whether to concatenate eye information into neural net.
  - use_headpose: Whether to concatenate headpose information into neural net.
  - number of training pics: The shape of the training set, its first element being the number of the training images.
  - number of test pics: The shape of the validation set, its first element being the number of the validation images.
  - net structure: Shape of the neural net layers.
  - train mean errors: The mean errors between predicted and actual labels when validated on training data.
  - valid mean errors: The mean errors between predicted and actual labels when validated on validation data.

  - train rmses: The root mean squared errors between predicted and actual labels when validated on training data.
  - valid rmses: The root mean squared errors between predicted and actual labels when validated on validation data.
  - best valid mean error: The lowest mean error the model was able to achieve in an epoch.
  - best valid rmse: The lowest mean squared error the model was able to achieve in an epoch.
  - valid prediction: The labels the model predicted with the lowest rmse while validating on this particular subject's validation data.
  - valid ground truth: The actual labels of the model for this particular subject.

2.7 Error messages

- "Currently not implemented."

  This message appears when calling a planned function that has not yet been implemented in the program. See: "Further Improvements" section in the Developer's Guide.

- "DB type not supported."

  This message appears when trying to use a database that's not in the list of supported databases.

- "Wrong path."

  This message appears when the path inputted by the user does not exist.

- "Inconsistent log file."

  This message appears during plotting when the log file inputted is of a different structure than the one expected.

- "Log can't contain both error types."

  This message appears during hyperparameter plotting when the log file inputted contains both rmse and mean error, and the program can't decide which one to use.

- "All subjects skipped. Not enough validation data to do crossvalidation."

  This message appears at the end of crossvalidation with the neural net, if there was not enough data for validation to occur at all.

2.8 Warning messages

- "No image for subject. Skipping subject."

  This message appears when our deep network tries to validate on a subject when no validation images are left for subject. It doesn't cause any problems, the program will just skip the subject.

- "didn't process image, because can't find all shape parameters:"

  This message appears during preprocessing, when one of the 49 facial keypoint markers isn't located inside the 2D mesh. It doesn't cause any problems, the program will just skip the image.

- "didn't process image, because of too big original rotation:"

  This message appears during preprocessing, when the image we are trying to preprocess has a too big original rotation. We'd like to skip these kind of images, because a too big original rotation can distort the image produced. It doesn't cause any problems, the program will just skip the image.

- "didn't process image, because descriptor documents not found:"

  This message appears during preprocessing, if the documents generated by Zface can't be found or are empty. It doesn't cause any problems, the program will just skip the image. (Zface generates empty descriptor files when it isn't able to locate the face on the image.)

3 Developer's Guide

3.1 Requirements

- **Problem:** Abstract: Our goal is to predict the gaze of human beings. Our input is a number of images about human beings, with their eyes at least partly open and their gaze visible, paired with the corresponding known gaze angles in the x and y dimensions, which we will have to transform to grayscale, resize, cut out the relevant parts from it, and perhaps use some additional transformation on it in order to obtain the final input data and labels, represented by matrices and vectors respectively. Our output will consist of angles in the x and y dimensions representing the estimated gaze angles in those dimensions. These will be compared to the known gaze angles corresponding to the same images in order to get a loss function which we want to minimize.
  More specifically, our tasks are the following:

  1. Augment and preprocess data from the ELTE and CAVE databases.
  2. Modify an existing Support Vector Regressor then optimize it with Bayesian optimization on a computer cluster.
  3. Create a new Deep Convolutional Neural Netowrk implementation and optimize it.
  4. Try both methods with personalized training data by further training the model with pictures of the subject on whom we will perform validation.
  5. Make plots and figures to help the presentation of the methods and the results. Compare the Support Vector Regressor with the Deep Network. Investigate the relationship between having information added to our models about the eye markers and headposes and the precision of the gaze estimation. Investigate the relationship between the number of personalization images and the precision of the gaze estimation. Draw conclusions (see: "Testing" and "Discussion").

- **Non-functional (quality) requirements**

  The program should be:

  - Easy to use by the target audience, with intuitive interface
  - Fast
  - Extensible

- **Functional requirements**

  The program will use:

    - Linux operating system
    - Python programming language
    - Command Line Interface
    - A main program used for presentation of the software's functionalities.

- **Planned solutions:**

    1. First of all we need obtain some data: it can either be in form of video or pictures and it should depict the face, particularly the eyes of human beings. The more subjects and the larger variance in head rotations we have, the better it is for our model's generalizational capabilities. It's important to know exactly where the subjects on the pictures are looking at because without this information ("labels"), we cannot do supervised learning.
    If our data is in the form of videos, we should cut out every n^{th} frame from them. If n is too small, the difference between consequent images will be small and it won't be worth the excess RAM usage during training. If, conversely, n is too large, we will fail to fully utilize our data.
    Data augmentation is a crucial step in the process. With a simple trick of reconstructing the face in 3D on each frame and rotating it, we can give our learning algorithm a lot of extra information which can boost its confidence. In this particular instance, we will rotate the faces in some predefined yaw/pitch/roll range. We can rotate them indefinitely many times. Again, there will be a trade-off between the amount of information we gain and the increased computer power we will have to utilize in order to process this excess data. We have to find a sweet spot.

    2. We already have a training algorithm utilizing a Support Vector Regressor as part of a larger project, provided by the NIPG group[3]. After some modifications, we can call its executable as a python subprocess with its parameters for crossvalidation. We have to specify a search space, then run our Bayesian Optimization on it. It will also generate the eye markers and headpose information we will later need for the training of our Neural Net.

    3. We will have to read our data into variables, resize them to the desired iod (which will be a hyperparameter), cut out the relevant parts (one of the eyes), and read the headpose and/or eye marker information from the files generated earlier. This is our training data. Finally we have to calculate the gazes associated with the eyes. These will be our labels.
    Now that we have our data and labels, we have to sort out which of these

will be in our training and validation set, and load the corresponding information for both sets. Finally, we should normalize our data in order to help speed up the convergence of our network.
We will give the appropriate data to our leave-one-out crossvalidation, which will go through every one of our subjects and validate on them using their own data and labels for validation and the rest of the data and labels for training. An exception from this is when we use personalization. We should put aside about twenty pictures per subject for personalization and use only the rest for validation. Out of these twenty personalization pictures, we will use the amount given as a hyperparameter as part of the training data.
The only thing left is to find out what architecture we want to use, then set the net's weights with a batch gradient descent algorithm, epoch by epoch. In order to be able to stop the model before it starts overfitting, we will have to implement a technique called "Early Stopping".
During the process we will want to log out our results.

4. To validate our model, we will use a technique called "Leave-one-out cross-validation". First, we divide our data into two groups: training and validation. We do this, in order to find out whether or not our model can be successful with data other than the ones it trained on. In other words, we want it to learn how well our model can learn the gaze direction of humans, not how well it can learn the noise in the training images. If we were to test our model on the same images as the ones we trained it on, it would be easy to attain 0% error, but it could still perform poorly on real data.
That said, we might want to learn the particularities of the subject we're currently testing on without learning all the noise of the training data. This means that our net will be attuned to our specific subject, but it won't be attuned to the specific training images.
It's not hard to imagine a scenario where this might be a useful feature. Imagine that we have an application that uses gaze detection. Before our user starts using our application, we can further train our already trained model on images taken of our user. This might take a little time, but it could also make our application much more precise for our user. Again, a trade-off we should investigate.

5. We will plot some results:

   (a) Heatmap of the headposes: The heatmap can be calculated by obtaining the yaw/pitch values for all the processed images in a given database, then converting the number of occurrences in a certain yaw/pitch region to colors from red (biggest) to blue (smallest).

(b) CNN train/valid convergence: Make a plot that depicts the convergence of our model on the training versus validation data, utilizing logged information about the root mean squared error of our model on those two groups of data epoch by epoch. This will present us with an intuitive picture about the rate of convergence and the amount of overtraining our model produced throughout its learning process.

(c) Plots of the logged hyperparameters: Make a scatter plot of every parameter the parameters being on the x-axis, and the error being on the y-axis, utilizing the logged hyperparameters and the error they gave. This will help us know more about the relationship between the parameters and the efficacy of our learning algorithm.

3.2 Problem Specification

1. **Raw Data** →preprocess.py →**Processed Data**

   - preprocess.rotate_images(data_folder, rots_per_pic):
     Generates rotated images and facial keypoint markers.
     - Description:
       Rotates images and produces the new coordinates for their facial keypoint markers.
       Creates rotated files and new markers in the same folder where the input files came from.
     - Inputs:
       data_folder : String
       Folder containing raw data to be processed.
       rots_per_pic : Integer
       Number of rotations to be produced per image.
     - Invariants:
       Input path should exist.
       Number of rotations should be a natural number.

2. **Processed Data** →hyperopt_search.py →**Logs**

   - hyperopt_search.svr(project_folder, logfile, db_type, err_type, evals, pers, mongo, port):
     Optimizes the free parameters of our Support Vector Regressor and logs the results.

- Description:
  Uses bayesian optimization to search for optimal hyperparameters.
  Logs the results in a log file.
- Inputs:
  project_folder : String
  Main folder of the project.
  logfile : String
  Desired name of file to log into.
  db_type: String
  Name of database to use.
  err_type: String
  Type of error to minimize.
  evals: Integer
  Number of evaluations.
  mongo: String
  Whether or not to run search on cluster (currently not implemented)
  port: Integer
  Port on which mongodb will run the cluster search (currently not implemented).
- Invariants:
  Project folder should exist.
  Database type should be one of the following: 'elte1'/'elte2'/'elte3'/'cave'.
  Error type should be one of the following: 'rmse'/'me'.
  Number of evaluations should be a positive integer.
  Mongo should be one of the following: 'y'/'n'.
  Port should be a free valid port number.

3. **Processed Data** →dcnn.py →**Logs**

- dcnn.get_results(project_folder, db_type, db_size, desired_iod, use_headpose, use_eyes, num_train_imgs, num_pers_imgs, lambda_l2, dropout1, dropout2, h1_neurons, h2_neurons, es_patience, batch_size, best_weights_list, use_pretrained_model):
  - Description:
    Read and process data, then perform leave-one-out crossvalidation on it.
  - Inputs:
    project_folder : String

Main folder of the project.
db_type: String
Name of database to use.
db_size: Integer
Number of subjects in the database.
desired_iod: Integer
Interocular distance. of the subjects in the resized images.
use_headpose: Boolean
Whether or not to concatenate headpose information to the net.
use_eyes: Boolean
Whether or not to concatenate eye marker information to the net.
num_train_imgs: Integer
Number of training images per subject.
num_pers_imgs: Integer
Number of personalization images per subject.
lambda_l2: Float
Amount of l2 regularization to use
dropout1: Float
Probability for dropout in the first dropout layer.
dropout2: Float
Probability for dropout in the second dropout layer.
h1_neurons: Integer
Number of neurons in the first hidden layer.
h2_neurons: Integer
Number of neurons in the second hidden layer.
es_patience: Integer
Number of epochs where the error increases before Early Stopping kicks in.
batch_size: Integer
Size of minibatch to use for the descent.
best_weights_list: List
List of weights the neural net obtained in a previous training.
use_pretrained_model: Boolean
Whether or no to use the weights of a previous model as initial weights.

- Invariants:
  Project folder should exist.
  Database type should be one of the following: 'elte1'/'elte2'/'elte3'/'cave'.

Database size should match the number of different subjects in the database given.
Desired interocular distance should be a multiple of 32.
Number of training images should be a natural number.
Number of personalization images should be a natural number.
Either number of training images or number of personalization images should be positive.
Lambda l2 should be 0 or positive.
Dropout probability should be between 0 and 1.
Hidden layer neurons should be a positive integer.
Early Stopping patience should be a positive integer.
Batch size should be a positive integer.
Best weights list should be a list of weights compatible with this neural net.

  - Outputs:
    best_weights_list : list
    List of weights produced by the net when validation error was the lowest.

4. **Processed Data + Logs** →plot.py →**Plots**

- plot.hyperparam(data_folder):
  - Description:
    Logs loss as a function of hyperparameters for every hyperparameter one by one.
  - Inputs:
    db_path : String
    Path of logfile.
  - Invariants:
    Path of logfile should be an existing file and should contain log in a particular format.
- plot.train_valid_convergence(data_folder):
  - Description:
    Logs training and validation convergence from a log file containing the training and validation rmse of a neural net epoch by epoch (CNN).
  - Inputs:
    db_path : String
    Path of logfile.

  - Invariants:
    Path of logfile should be an existing file and should contain log in a particular format.

- plot.heatmap(data_folder):
  - Description:
    Makes a heatmap from the yaw/pitch degrees of the rotated images contained in input path.
  - Inputs:
    db_path : String
    Path of database.
  - Invariants:
    Path of database should be an existing path.

3.3 Structural and behavioral diagrams

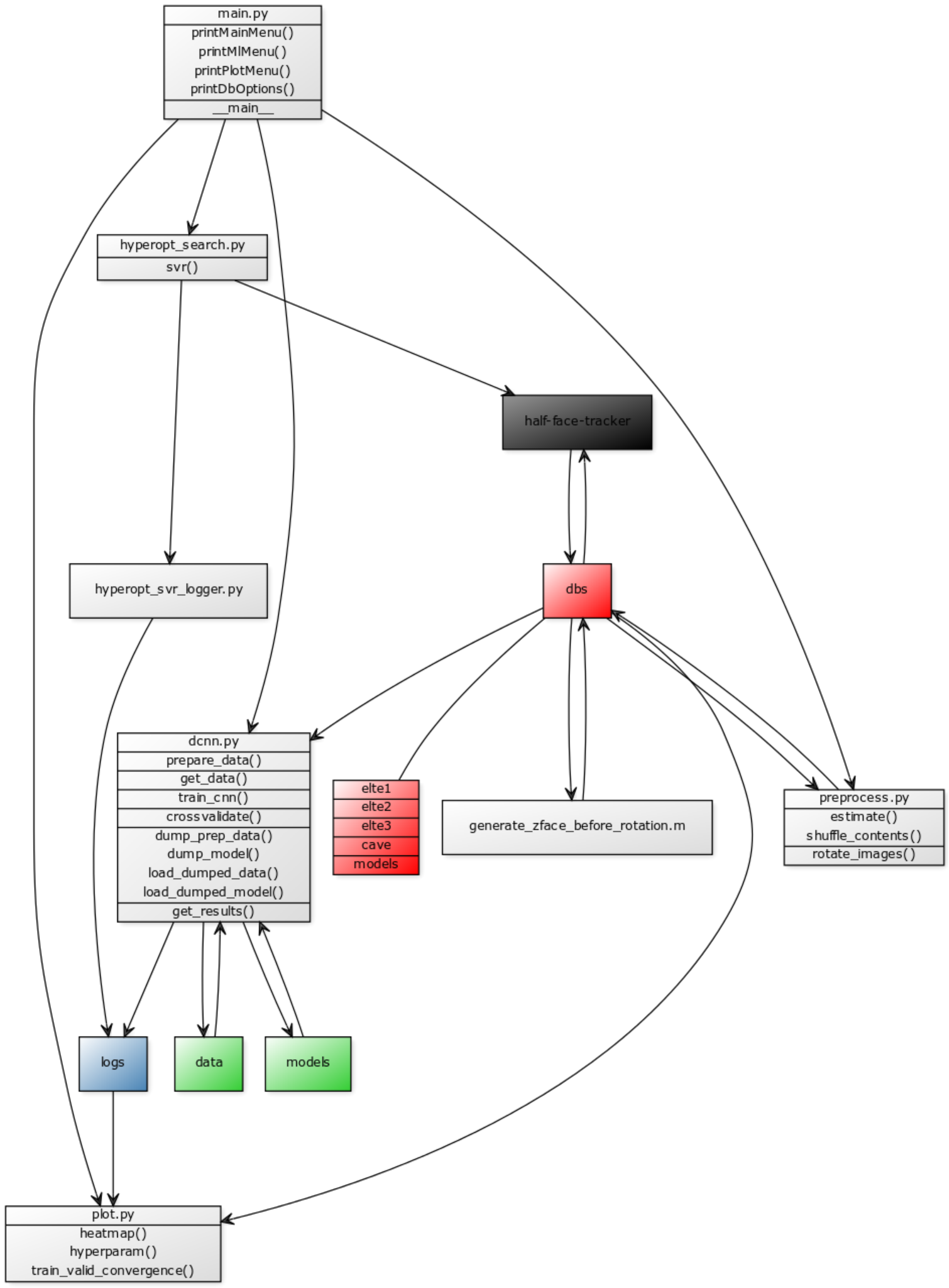

Figure 3: Structural Diagram. The structural diagram shows the static structure of the program. It enlists the modules, their methods and the hierarchical relationship between them and the rest of the files in the project folder.

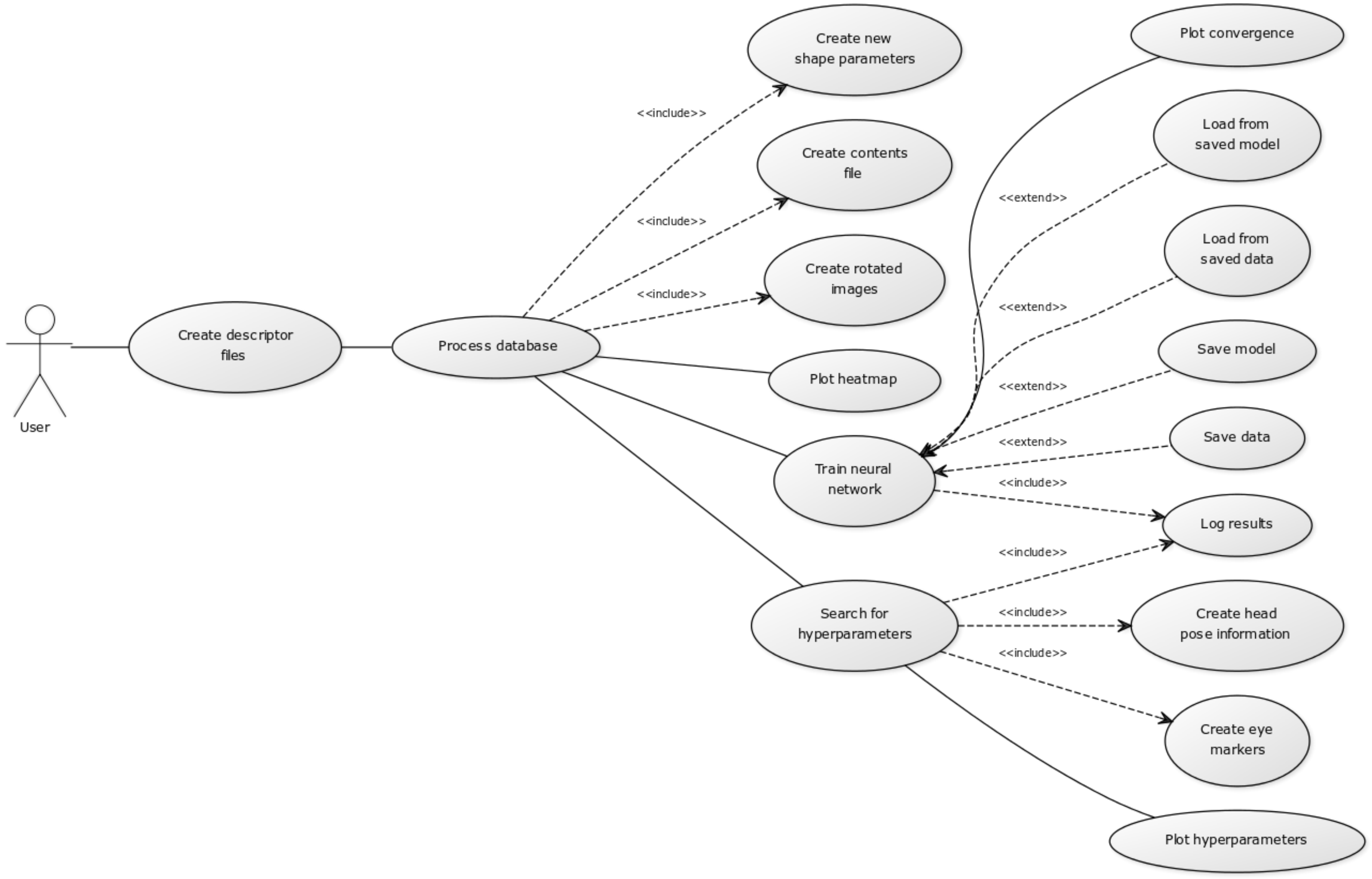

Figure 4: Behavioral Diagram. The Behavioral diagram shows the dynamic structure of the program from the viewpoint of the user. It highlights the changes that occur when using certain modules and the chronological order in which they should be used.

3.4 Planning and Implementation

1. Augmenting our data
   First we have to find a way to generate 3D information about 2D images. The solution for our problem is Zface, a software which uses a 3D cascade regression approach. From a 2D image of a person's face (our raw data) it registers a dense 3D shape (mesh). The algorithm works by first estimating the location of a dense set of markers and their visibility, then reconstructing face shapes by fitting a partbased 3D model[14].
   Zface is a collection of Matlab libraries. After modifying some of its source code to fit our needs, we ran the program. This generated our descriptor files next to our raw data. This will not be a part of the main script used for the presentation of the program.
   The next step is to load our raw images and the descriptor files generated, rotate the images, then save the images produced.
   We decided to use an iterative approach to search for all the subfolders inside our database (which is given as the first argument).
   The next argument of our function is the number of times we want to rotate each image. In order to produce exactly the given amount of rotations, we count the number of previous rotations, or more precisely, the files with the same beginning as our current file, but followed by the string "_processed_". We then count the occurrences of these files for all images, making a dictionary of the filenames and their respective number of rotated counterparts. This way, we can always come back and do more rotations on a given dataset, or continue a previously interrupted process without having to delete the progress made so far.
   We decided to only process files with initial rotation between a yaw of $-30°$ - $+30°$ and a pitch of $-15°$ - $+15°$. This choice was made in order to make sure we don't produce too much distortion in our final product.
   Zface is sometimes unable to generate the descriptor files necessary, so we had to test for whether or not the descriptor files are empty. If they are, we will leave the images associated with them untouched as well.
   Now to begin our loop: First of all, we load up our current image and its descriptor files, and extract the information we need from them.
   Then we rotate, scale and translate our 3D mesh from the absolute coordinates to the relative coordinates, in order to be able to fit the 2D surface (face) on top of it.
   Now we scale and translate it back to absolute coordinates, and rotate the mesh with the face into yaw=pitch=roll=0 by extracting out the information

about its initial rotation and reversing it. After this, we can apply our random rotations, with the help of a random number generator.
The only thing left to do now is to scale it, rotate it back to relative coordinates, and project it to 2D with orthographic projection, finally to warp the original image to the new surface with the help of Skimage's "PiecewiseAffineTransformation" and "warp" function.
We also have to generate the facial keypoint markers for our newly processed images. Unfortunately, we cannot use Zface again, because the markers it would generate would be highly skewed as a result of the artificial nature of the processed images. Not to worry though, because we can modify our original technique for rotating images to rotate the markers too. We originally used Skimage's piecewise affine transformation in order to wrap our original image to the 2D orthographic projection of our rotated 3D mesh. This method uses Delaunay triangulation to divide the original mesh into many little triangles, then transform those triangles and the pixel information contained within them into the new surface, finally interpolate the resulting triangular surfaces to obtain the final, warped image. The modification we add to this algorithm to fit our purposes is to rewrite Skimage's "estimate" function, and make it return the produced triangles to us. Now, by calling matplotlib's "path" module, we have to look for the 49 facial keypoint markers inside the triangles of the source mesh and extract out the transformation matrix associated with every one of them. (If for some reason, one of the markers is outside the original surface, we skip that image and print the problem on the screen.) After finishing with this, the only thing left to do is to perform our transformation by applying the given transformation matrix. This gives us our new coordinates, which we can save, giving it the extension "_0.txt" after the name of the transformed image we generated the markers for. This will later be used by "half-face-tracker".
It's wiser to shuffle the data a little bit before giving it to our learning algorithms. We call the function called "shuffle_contents" which does precisely that. It takes the contents of the "contents" file generated in the last step and randomizes the order in which the images per subject and the rotations per image follow one another, but leaving all the images that are taken of the same subject and all the rotations of the same image next to each other.
Finally, we list some statistics about the process.

2. Optimizing the Support Vector Regressor
   First of all, we had to modify to the software inside the folder "gaze_svm" to fit our needs. The software contains a support vector machine and capabilities

for leave-one-out crossvalidation. We will not go into detail about the nature of the modifications and the inner workings of this software, because it would unnecessarily complicate things, and it is not strictly part of this thesis. All we need to know is by calling the executable "half_face_tracker" and giving it the following arguments: "[IOD] [feature type (1: hog, 2: lbp)] [feature param] [train imgs] [test imgs] [pers imgs] [train rots] [test rots] [pers rots] [eps] [C] [p] [db: 'elte1'/'elte2'/'elte3'/'cave'] [use headpose] [use eye markers]", we can extract some useful results from the information printed on the screen.

What to do with this information? After calling "half-face-tracker" with parameters sampled from the search space as a subprocess, we can extract out the average rmse or mean error (depending on which one we want to minimize) with the help of regular expressions, then give it to Hyperopt (a Python library for serial and parallel optimization over awkward search spaces[15]) as the output of an unknown function to minimize.

We also have to define a search space: The possible values and optimal distributions it should use for sampling.

The algorithm we want to use for this task is TPE (Tree Parzen Estimator), which is a form of Bayesian optimization. After defining the search space, the objective to minimize and the suggested algorithm, we can call hyperopt's "fmin" function in order to start the search for optimal hyperparameters. During this process we want to print out some information about the inner workings of the program to be able to see what's going on, and log the results with the help of a logger script run as a subprocess.

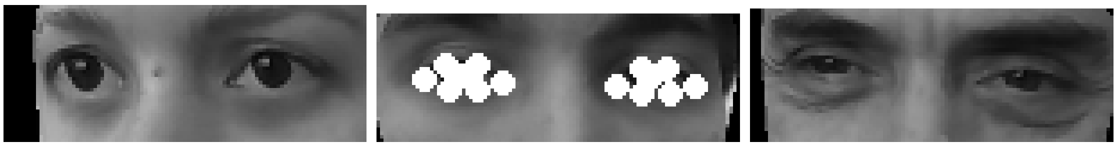

Figure 5: Examples of training images for the SVM. The eye markers are highlighted on the middle one.

3. Constructing and training a deep convolutional neural network

   First of all, we load the paths to our images from the "contents" file, load the images as grayscale and the descriptor files, headpose and eye marker information associated with the images. If some information is missing, we skip the whole thing.

   We resize the images so that the interocular distance of the subjects will become the desired interocular distance specified in the parameters, then crop them to a predefined size around the center of the left eye. We can omit the information about the right eye, as it doesn't give us much more additional

information. (At the distances the subjects are located from the camera the difference between the gaze of the left and right eye is negligible.) This will be our training data. We can calculate the gazes from the information about the initial gaze and the rotations. The results of the calculations will be our labels. We pass the training data, labels, headpose and eye marker information alongside the path associated with them to our next function.
The next function will take this data and depending on whether we want to create a training or validation set and the amount of training and personalization images we want to use, it will give us a subset of the data. It also reshapes and normalizes the data so the neural net will be able to read them, and will have an easier time converging on them.
This processed data will be the input of our neural net. We will load the data into Theano's symbolic variables which will always represent a mini-batch of training data and will change dynamically during training. We will then start the standard training loop, updating the model with a mini-batch amount of data at every step. For the gradient descent we use Adamax as our loss function. For the sake of Early Stopping and the monitoring of our progress we also evalute the loss produced on our validation set as well as on the training set. It's important to turn off dropout here, which we accomplish by setting the "deterministic" parameter of our validational loss function to "False". We also want to apply optional l2 regularization to our loss function to help reduce overfitting. Theano has a function which computes it for us, we just have to multiply it by the hyperparameter l2_lambda. λ can reduce or increase the influence of the regularization. After computing the regularization term, it gets added to our training loss function. Now that we defined all of our expressions, we can start the training loop. This loop will continue until it either reaches the maximum number of epochs, or until it can't improve its validation performance any further after a certain number of epochs. After it stops, it will save its best weights and the training as well as the validation loss produced with those weights. It will also give the best weights produced during training as one of its outputs.
Leave-one-out crossvalidation is the technique we are using to test the performance of the model on the whole database, training our net on one subject at a time, iterating the process described in earlier paragraphs subject by subject, changing the contents of training and validation data according to the subject we're validating on. At the end of the crossvalidation we print the average losses produced by the individual validations among other things.
Of course, during the whole procedure, we print out and log into files all the relevant information, so the whole process will be documented in real time as

well as stashed for later use.

We also included capabilities to save and load the weights of previous models and the processed databases as serialized data. These will be stored in the folders "models" and "data" respectively.

The neural net uses an architecture similar to the one presented in the following paper:[16]. One particularity of this model is its usage of concatenation layers, which can be turned on or off depending on whether we want to use the headpose and/or eye marker information generated by our "half-face-tracker" software or neither of them. See Figure 7 for a semantic drawing of the network.

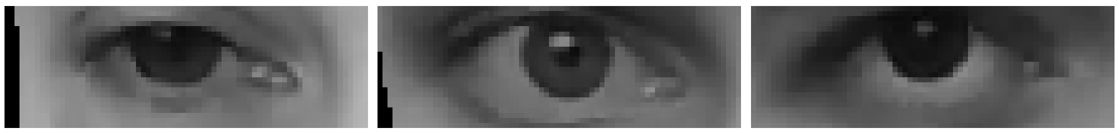
Figure 6: Examples of training images after converting to grayscale, resizing and cropping for the network.

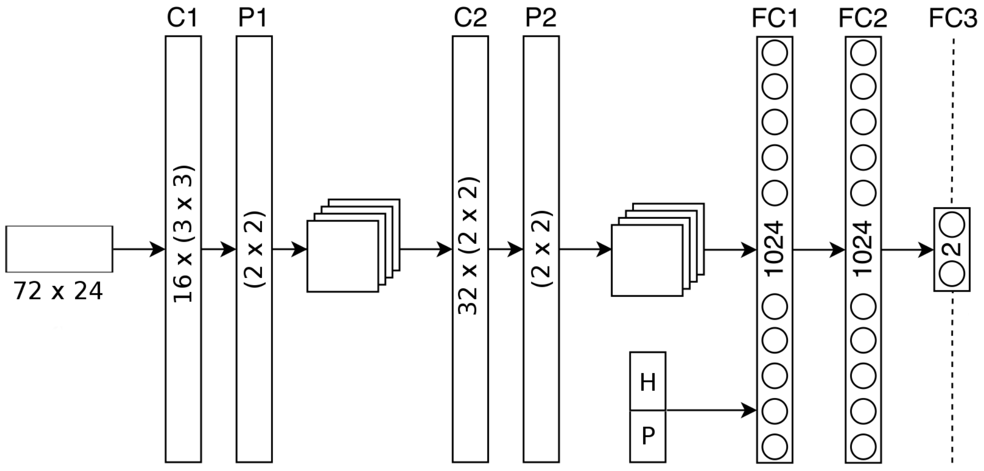

Figure 7: Architecture of our deep net. Two convolutional layers (C1,C2) each followed by a pooling layer (P1,P2) (number and size of filters denoted on the figure), the output of which optionally gets concatenated with headpose and eye marker information (H,P) and transformed through two fully connected layers (FC1,FC2) each containing 1024 neurons, finally narrowed down to the two output gazes (x and y) by the two neurons of the third fully connected layer (FC3). The network also contains two dropout layers, one of them after the second pooling and the other after the first fully connected layer.

4. Plotting results

   (a) Heatmap of the headposes:
   First we extract the paths of our processed images and the pitch and yaw values from those paths.
   The next step is to use Numpy's "histogram2d" function to make a histogram with 120 bins from $-60°$ to $60°$ pitch and yaw (in order to show the contrast between our processed range and its environment), then use the histogram in Matplotlib's "imshow" function, which converts it into a heatmap. For a smooth finish, we are using bicubic interpolation between the bins.
   Finally, we label the x and y axis and the plot, then display it.

   (b) CNN train/valid convergence:
   By extracting information about the training and validation losses previously logged by our deep convolutional neural network, we can use matplotlib to draw lines between our losses as a function of epochs, the losses being on the y-axis and the number of epochs on the x-axis. Now we can label and display our data.

   (c) Plots of the logged hyperparameters:
   We use regular expressions to find our hyperparameters in the log file, then store the value of the given variables as floating point numbers in Numpy arrays. Finally, we make one scatter plot per hyperparameter, the hyperparameter being on the x-axis and the losses we got as a function of that parameter being on the y-axis. We display the plots produced.

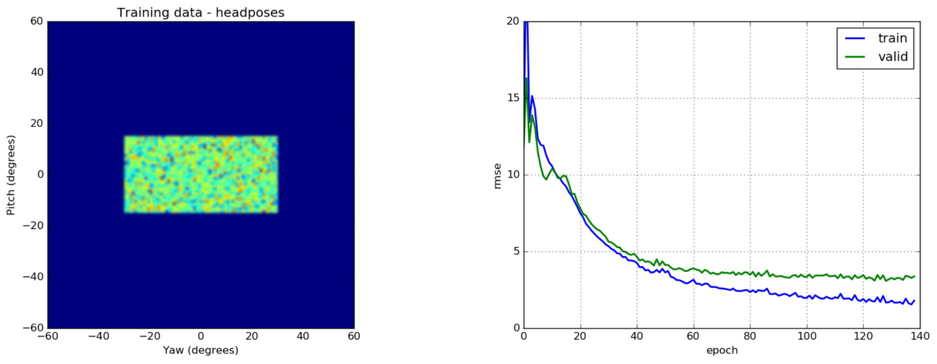

Figure 8: Example of heatmap of headpose distribution on the left, and convergence plot on the right

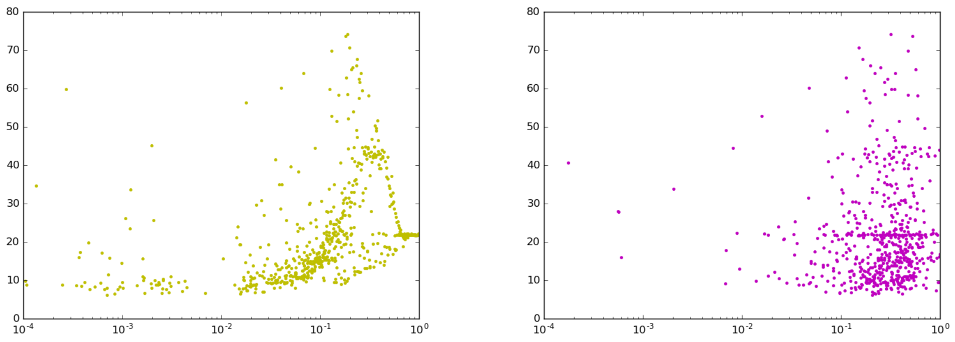

Figure 9: Example of RMSE in function of the p and c parameter of the SVM obtained from hyperoptimization

3.5 Testing

The software has been tested in multiple ways during the time it was used to help create numerical results in a paper published by the Neural Information Processing Group. During this project, it generated the processed data for two different databases: "ELTE3" and "CAVE". A bayesian optimization was then started to find the optimal hyperparamters of the support vector machine, and a manual optimization for the deep convolutional neural network. Finally, the algorithms with their set default parameters were run in multiple configurations, changing the values of some key variables investigated: the number of personalization images and the use of additional techniques, like headpose and eye marker estimation. The goal was to find out how much influence these key parameters had on the results. [Table 1] [Table 2]

The main script used for presentation was also tested for erroneous inputs, including bad datatypes (integer instead of string, float instead of integer, etc..), misspellings and inconsistent log files.

Features	32IOD	96IOD	32IOD	96IOD	32IOD	96IOD	32IOD	96IOD
None	6.89	7.11	6.69	7.06	5.98	5.06	10.37	8.62
Eyes	4.88	7.69	5.20	5.36	5.64	5.07	10.11	8.53
Headpose	6.17	6.35	6.05	6.06	3.92	3.85	8.07	6.97
Eyes+Hp	3.78	7.96	5.07	5.27	3.82	3.86	8.28	7.12

Table 1: Mean errors produced by not adding adding any extra information to the net, then by adding information about either the eye markers or headpose, finally both.

Pers.	ELTE	ELTE3 CNN 32IOD		ELTE3 CNN 96IOD		CAVE CNN 96IOD	
images	SVM	(a)	(b)	(a)	(b)	(a)	(b)
0	3.78	3.82	3.83	3.86	3.89	7.12	7.09
5	2.81	2.91	3.06	2.45	3.29	6.13	6.47
10	2.56	2.61	2.75	2.24	3.24	5.59	6.27
15	2.36	2.34	2.26	2.02	2.28	4.98	5.33
20	2.22	2.21	2.04	1.80	1.93	4.61	4.59

Table 2: Mean errors produced by adding different number of personalization images to the training set. (a) Training net with personalization images in the training set from the start. (b) Training net with no personalization images, saving its weights then further training the net with only personalization images.

3.6 Discussion

In this thesis we managed to present all the necessary steps involved in a machine learning task starting with some simple raw pieces of data. We implemented a novel data augmentation technique devised by the Neural Information Processing Group[3] along with an optimization technique for an existing learning algorithm and a brand new convolutional neural net designed for this task. We have written about this process in detail with the help of plots and figures. We have also reached some conclusions in light of the results:

- Both the Deep Convolutional Neural Network and the Support Vector Machine managed to predict gazes with decent accuracy, with mean errors as low as an average 2°, crossvalidating on the "ELTE3" database.
- Although giving information about the position of eye markers to the convolutional neural net doesn't seem to have a big effect on its efficacy, information about the head pose does. When teaching the Support Vector Machine both pieces of information seem to produce a notable effect.
- The act of introducing even a small amount of personalization images in the training data can have a dramatic effect on the final prediction.

3.7 Further Improvements

In the event of wanting to run a large scale study with more power (a computer cluster) and more time on our hands, we should make two simple modifications to our code:

- Extend automatic hyperoptimization to work with the Deep Convolutional Neural Network.
- Modify Hyperopt to work on a computer cluster with the help of MongoDB. Instructions:[17]

4 References

[1] Phil Simon (March 18, 2013). Too Big to Ignore: The Business Case for Big Data. Wiley. p. 89. ISBN 978-1-118-63817-0.

[2] KI Conference, `http://ki2016.org/` (Accessed 25 May 2016)

[3] Neural Information Processing Group, `http://nipg.inf.elte.hu/` (Accessed 25 May 2016)

[4] Coursera Machine Learning, `https://www.coursera.org/learn/machine-learning` (Accessed 25 May 2016),
Instructor: Andrew Ng

[5] Stanford CS231n, `http://cs231n.github.io/` (Accessed 25 May 2016),
Instructor: Andrej Karpathy

[6] Coursera Neural Networks, `https://www.coursera.org/course/neuralnets` (Accessed 25 May 2016),
Instructor: Geoffrey Hinton

[7] Patrick Winston. 6.034 Artificial Intelligence, Fall 2010. (Massachusetts Institute of Technology: MIT OpenCourseWare), http://ocw.mit.edu (Accessed 25 May, 2016). License: Creative Commons BY-NC-SA

[8] LIBSVM, `https://www.csie.ntu.edu.tw/~cjlin/libsvm/` (Accessed 25 May 2016)

[9] Backpropagation algorithm, `https://en.wikipedia.org/wiki/Backpropagation` (Accessed 25 May 2016)

[10] Theano, `http://deeplearning.net/software/theano/index.html` (Accessed 25 May 2016)

[11] Nvidia CUDA, `https://developer.nvidia.com/cuda-zone` (Accessed 25 May 2016)

[12] Configuration of Theano, `http://deeplearning.net/software/theano/library/config.html` (Accessed 25 May 2016)

[13] "Gaze Locking: Passive Eye Contact Detection for Human–Object Interaction," B.A. Smith, Q. Yin, S.K. Feiner and S.K. Nayar,
ACM Symposium on User Interface Software and Technology (UIST),
pp. 271-280, Oct. 2013.

[14] Zface, http://www.pitt.edu/~jeffcohn/biblio/Jeni15FG_ZFace.pdf (Accessed 25 May 2016)

[15] Hyperopt, https://github.com/hyperopt/hyperopt (Accessed 25 May 2016)

[16] Appearance-Based Gaze Estimation in the Wild: http://arxiv.org/abs/1504.02863 (Accessed 25 May 2016),
Xucong Zhang, Yusuke Sugano, Mario Fritz, Andreas Bulling

[17] Instructions on parallelization, https://github.com/hyperopt/hyperopt/wiki/Parallelizing-Evaluations-During-Search-via-MongoDB (Accessed 25 May 2016)