Capstone Project

Machine Learning Engineer Nanodegree

Roman Semenyk
June 5th, 2017

Definition

Project overview

What is an ideal Human Computer Interaction (HCI) interface? The ultimate HCI interface is as natural and easy as possible. It aims the user thoughts and desires to be are directly understood by computers. Before this future comes true, we experiment with available alternatives. Personal assistants[1], gesture and emotion recognition systems[2], gaze trackers[3] are a few among the many examples of such emerging interfaces. They allow more people to interact with computers easily.

In this project, I created an application that acts similarly to a computer mouse or a touchpad. The application is using an ordinary web camera (usually built into the device). No calibration or hardware configuration is necessary. User's head movements are translated into pointer movements. The user's eye states (opened/closed) are interpreted as mouse button states (released/pressed). The application can be useful for people, who cannot or don't want to use mouse or touchpad, people with disabilities. Gaming and entertainment are the use cases as well. The application is using a classifier to detect open/closed eye state and is tolerant to the changes in light conditions and user-to-camera distance. The classifier is trained on publicly available data, which will be described in the next sections.

Problem Statement

The goal of the project is to create a python application, which maps the user's head movements into the mouse pointer movements and user's eye blinks into mouse buttons clicks. Emphasis is made on training a reliable eye-state binary classifier, as it is the key machine learning problem of the project.

The steps to reach the goal are as following:

---

[1] R. K. Moore "Is spoken language all-or-nothing? Implications for future speech-based human-machine interaction" *arXiv preprint arXiv:1607.05174* (2016)
[2] S. Piana, A. Staglianò, F. Odone, A. Verri, A. Camurri "Real-time Automatic Emotion Recognition from Body Gestures", *arXiv preprint arXiv:1402.5047* (2014)
[3] S. Tripathi, B. Guenter "A Statistical Approach to Continuous Self-Calibrating Eye Gaze Tracking for Head-Mounted Virtual Reality Systems" *arXiv preprint arXiv:1612.06919* (2016)

1. Download and preprocess the dataset.
2. Train classifier that can determine if an input image is a closed or open eye.
3. Create an application that uses default webcam and detects user face with eye regions.
4. Make the app to track user face and eye regions, translate movement into the mouse pointer movement.
5. Use previously trained classifier to detect the eye states and blinks.
6. Filter natural eye blinks from intended blinks, map eye states to mouse buttons.

The final python application should be considered as a proof of concept and results will be used for further explorations and optimizations.

Metrics

Among all the metrics, ones that affect the user's experience most are the **accuracy** and **processing delay**. For the classifier, the common accuracy metric will be used to measure its performance.

$$accuracy = \frac{true\ positives + true\ negatives}{dataset\ size}$$

, where

true positives – correctly classified open eyes;

true negatives – correctly classified closed eyes.

Processing delay is the time between an input image (frame) is grabbed from the web camera till the application produces a result – pointer move and/or a button click. Optimal comfort framerate for a user is 30 frames/second. So, the target delay needs to be < 33 milliseconds.

Analysis

Data exploration

The publicly available dataset "Closed Eyes In The Wild"[4] (CEW) is used to train the classifier. This dataset consists of 24x24 pixel grayscale centered images of human eye regions. Images initially organized in 4 categories: "openLeftEyes", "openRightEyes", "closedLeftEyes", "closedRightEyes". For our classification task only the eye state is relevant, so all images will be combined into 2 classes "open eyes", "closed eyes". The images were taken under various light conditions, head and camera positions. Typical examples of both classes presented on fig. 1

[4] F.Song, X.Tan, X.Liu and S.Chen, "Eyes Closeness Detection from Still Images with Multi-scale Histograms of Principal Oriented Gradients" (2014)

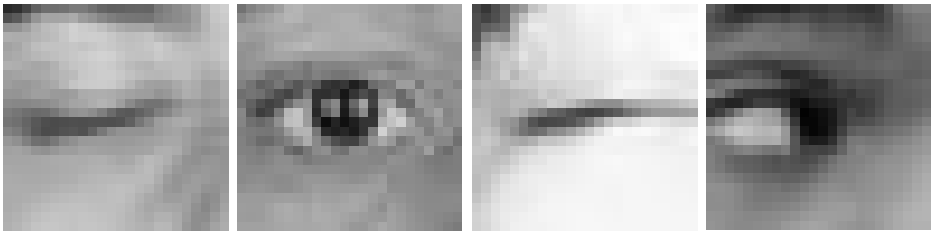


Fig. 1 Images from the CEW dataset

The dataset has about 2400[5] images for each class, so it is well balanced for binary classification.

Among the available data, there are few (less than 2-3%) blurry, low-contrast or exceptionally unclear images that could be considered as abnormal or outliers. This might happen due to low-quality source picture and upscaling to 24x24 pixels.

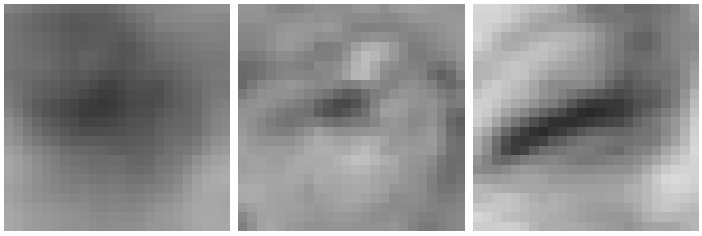


Fig. 2 Example "hard" cases – all images marked as open eye

They might be quite hard to classify correctly, however it is decided to leave them in the dataset as our goal is to create a reliable classifier that will take input from any ordinary web cam and under varying conditions.

Exploratory visualization

As images taken in different light conditions, let's see the distribution in average brightness of the pictures of each class. This visualization is relevant, as image mean brightness and its standard deviation will be used for data transformation described in Data Preprocessing section.

[5] Precisely 2386 and 2462 images of closed and open eyes respectively

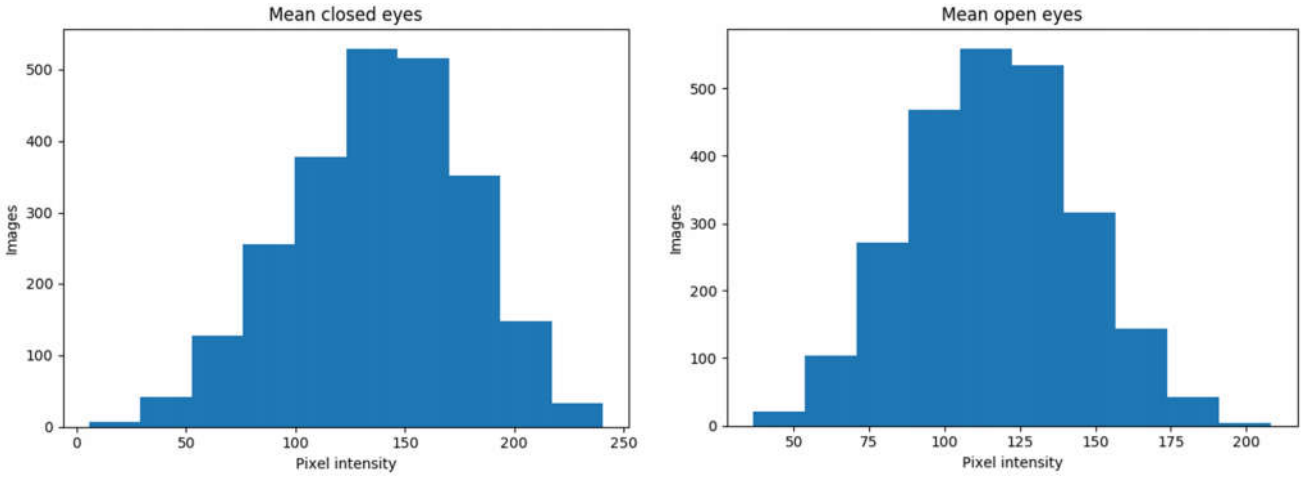


Fig. 3 Mean brightness of closed and open eye images

From the fig. 3 it is clear, that the distribution of brightness is normal for both classes. Images for closed eyes are slightly brighter in average and there are few very dark and very bright examples among them.

Algorithms and Techniques

The problem is an image classification task. As a solution, a Convolutional Neural Network (CNN) classifier will be used. This is the state-of-the-art algorithm for most image processing tasks. The algorithm outputs an estimated probability for each class (opened/closed eye, in our case).

The following parameters can be tuned to optimize the classifier:

- Preprocessing parameters:
  - Data augmentation parameters (see the Data Preprocessing section)

- CNN architecture:
  - Depth of the network (number of layers)
  - Layer types (convolutional, pooling, ReLu, fully-connected)
  - Layer parameters (see links above)

- Training parameters:
  - Training length (number of epochs)
  - Learning rate (how fast to learn)
  - Weight decay (prevents extreme weight values)
  - Batch size (how many images to process in a single step)
  - Solver type (what algorithm to use for learning)
  - Momentum (takes the previous learning step into account when calculating the next one)

The initial dataset images augmented with randomized scale and shift transformations in order to train a scale/position tolerant classifier. Zero mean and unit variance transformations also will be applied on resulting set.

The training phase is performed on GPU to speed up the process. In contrast, the inference is done on CPU to ease hardware requirements of the entire application.

Benchmark Model

A "classic" CNN called LeNet is used as a reference model. It was originally designed to classify hand-written digits of the MNIST database. With minimal modifications, a variant of this network is used as a benchmark for the researched problem. This allows using the same evaluation metrics to compare performance of the reference and final models. Benchmark model result will be highlighted in the Refinement section.

Beside of that, I set some minimal acceptable values for the selected metrics:

- Classification accuracy above 95%. As classification result is mapped to mouse clicks in some way, any noticeable misclassification will be annoying for the user.
- Processing delay below 33 milliseconds. As this is a user input application, it should not draw all the CPU power of the device. There should be enough resources for other applications user interacts with.

Methodology

Data preprocessing

The "data_preparation" notebook aggregates all the performed preprocessing and consists of the following steps:

1. Verify that the initial dataset is available, create folders for the augmented dataset. Delete any previously existing augmented data.
2. Set the augmentation settings and constants (discussed below).
3. Walk through the initial dataset and create a batch of augmented images for each input image.
4. Split the augmented images randomly into training and test sets.
5. Compute image mean for the entire augmented dataset and pack images into the format, best suitable for the chosen deep learning framework (Caffe, LMDB format, the tools shipped with the framework are used to perform this transformation).

The reasons, why image augmentation is beneficial for this project are summarized below:

- Initial dataset is centered on user eye pupil. For a real-time application, it is really challenging to track user eye pupil with high precision on each frame. It is easier to track eye region on the image, so the eye center can be offset from the tracked region center.
- Classifier should handle different eye size, as the distance from user's eye to the camera is not constant.
- Number of training examples is significantly increased with augmented images. CNN usually performs better on more data.

Image augmentation consists of the following settings and steps:

1. Mean and standard deviation computed for the initial input image.
2. New 32x32 pixel grayscale image is created and filled with random noise. Noise mean and std. deviation are equal to the values calculated in step 1.
3. Initial 24x24 pixel image randomly scaled between "scaleRange" setting values and projected on the new 32x32 image. Projection location is also random and controlled by the "shiftRange" setting. For examples, refer to the figure 4.
4. Steps 2 and 3 repeated "shiftCount" x "scalesCount" x 2 (initial image flip) times. So for each one input image, a batch of new different images is created. All augmented images in a batch get the class label from parent image.
5. Zero-mean and unit-variance normalizations are applied to the augmented images (is done during classifier training phase, in place).

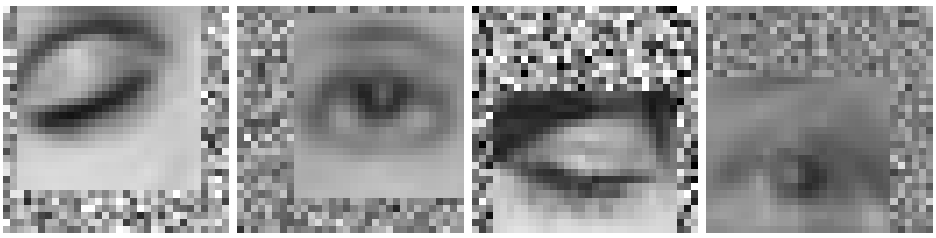


Fig. 4 Images from augmented dataset

During inference, grayscale images grabbed from the web camera video feed, eye regions extracted and scaled to 32x32 pixels. Similarly, zero mean and unit variance normalization is applied.

Implementation

Application development consists of 2 phases:

1. Classifier training (see the "classifier_training" notebook).
2. Whole application development (see the "program" folder).

In the first phase, classifier was created and trained on the preprocessed data, Caffe deep learning framework used for training the classifier. All the experiments, benchmark model tests and parameter fine tuning took place in this phase. In the notebook, actions executed (with extended comments) in following order:

1. Setup, link necessary modules. Set GPU mode for Caffe.
2. Define helper functions to train and test CNNs.
3. Define and train the benchmark model (architecture and parameters, training length).
4. Define, train and save the optimized model. The optimized model is a result of fine tuning of architecture and parameters of the benchmark model.
5. Visualize model training results. Plot training chart, confusion matrix, draw typical misclassified examples.

The final, optimized model NN architecture and layer parameters described below:

1. Input is Data layer; batch size is 100. This layer also performs zero mean and unit variance normalization.
2. Conv1 is a convolution layer with kernel size 6 and stride 1. The layer is followed by the ReLu activation layer (in place).
3. Pool1 is a max-pooling layer with kernel size 4 and stride 2.
4. Layers conv2, relu2, pool2 are of the same type as conv1, relu1 and pool1, but with different parameters, as shown on figure 6.
5. Fc1 is a fully-connected layer with 600 neurons, followed by another ReLu.
6. Score is a raw output layer, the inner product of the fc1 results.
7. Loss is a SoftMaxWithLoss service layer, that is used in training. The loss function is the [multinomial logistic loss](#).

Best learning results achieved with Nesterov's accelerated gradient[6] solver and about 3000 training epochs. Please refer to the "classifier_training" notebook for additional solver parameters.

To use the model in real application, input data layer batch size changed to 2 (for left and right eye on a frame), SoftMaxWithLoss layer replaced with Softmax. Prepared model is saved as "*model_deploy.prototxt*" file.

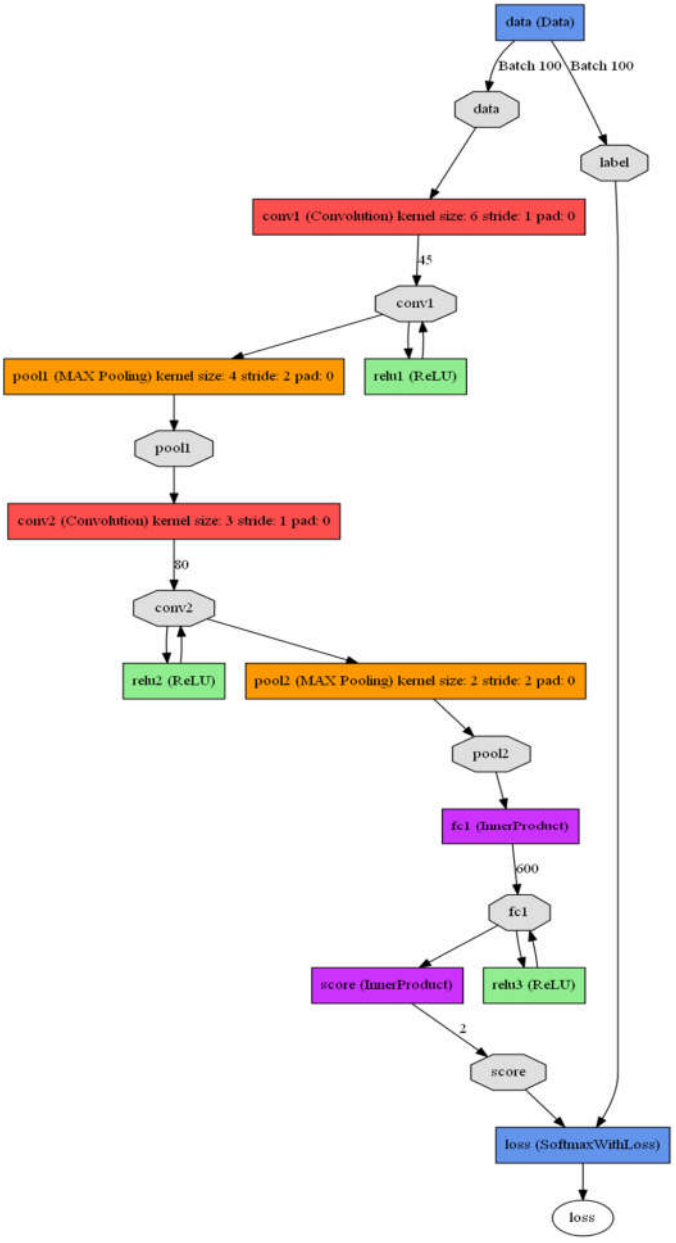


Fig. 6 Optimized model architecture

[6] Y. Nesterov. A Method of Solving a Convex Programming Problem with Convergence Rate O(1/√k). Soviet Mathematics Doklady, 1983

In the second phase, the python application was implemented. Its components and data flow are presented in figure 7. Open source libraries OpenCV and D-Lib used for image processing and other image-related tasks. For cross-platform mouse operations, the PyInput framework is used.

1. Default device web-camera is activated and the video-feed is grabbed frame by frame. User's face is expected to be clearly visible for the camera. Every image from the camera is gray-scaled and flipped. (see file "main.py")

2. Gray image is forwarded to the face detection and eye region tracking module (file "faceDetector.py"). If face is located in the frame, its coordinates (tip of the nose landmark) and estimated eye regions extracted. Very fast, state-of-the-art face landmarks detection algorithm[7] used to extract eye regions.

3. Eye regions preprocessed and provided as an input to the pre-trained classifier from the first phase (file "blinkDetector.py"). Eye state detection results and user's face nose tip point coordinates forwarded to the movement and blink analysis module (file "motionAndBlinkAnalyzer.py"). It subtracts coordinates from the previous frame to get the delta between two consequent frames.

4. Then the delta position results are filtered and smoothed to reduce jitter and to get the desired mouse pointer sensitivity. Thresholds and timing restrictions are applied to the classification results to reduce the number of false positive detections and to filter out natural blinks. (file "motionAndBlinkAnalyzer.py").

5. Mouse actions module gets mouse position change and state (opened/closed) for each eye. Then it moves the mouse pointer accordingly and assigns desired mouse button actions.

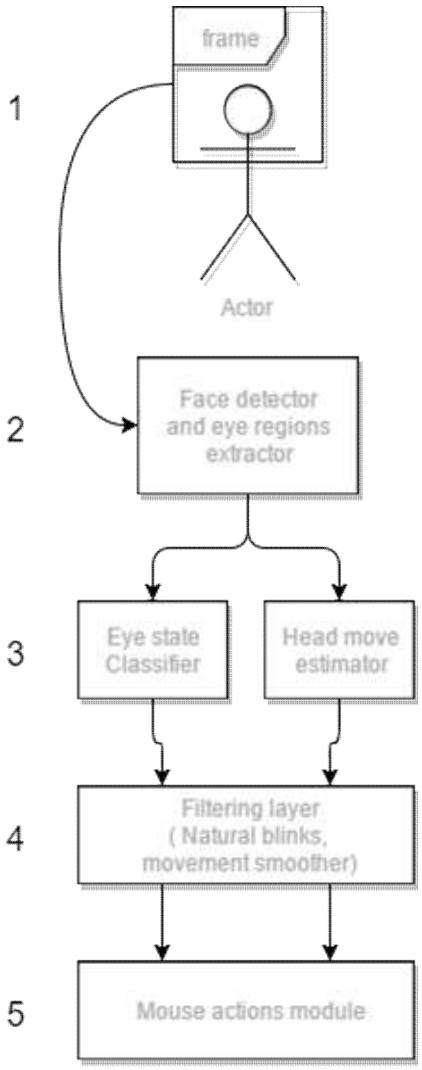


Fig. 7 Application components and data flow

Refinement

Benchmark model, the LeNet architecture with default learning parameters achieved the accuracy of about 93.8%. The final optimized model reached the accuracy of 97.2%. The score was improved using the following techniques:

[7] *Vahid Kazemi, Josephine Sullivan*; "One Millisecond Face Alignment with an Ensemble of Regression Trees" The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2014, pp. 1867-1874

- Adding ReLu layers after convolutions in NN architecture. This introduces some more non-linearity to the model.
- Adjusting parameters convolution and pooling layers, the number of learned filters.
- Configuring solver and learning parameters. Caffe NesterovSolver implementation showed better results and faster convergence.

The model was optimized iteratively by training with different parameters and comparing the resulting accuracy and train loss.

Figure 8 represents train & test loss and test accuracy dynamics over learning iterations. These charts were used extensively to compare different model variants.

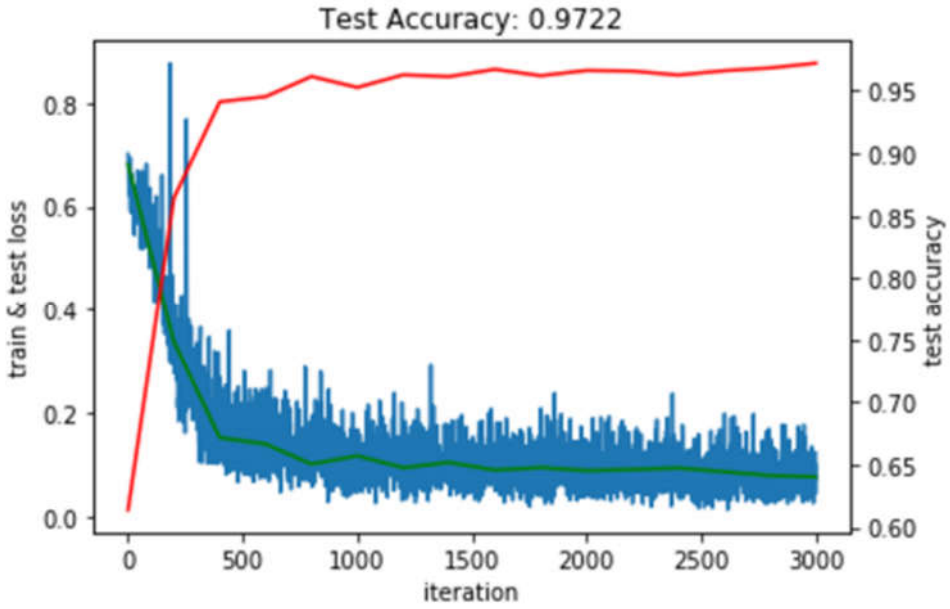


Fig. 8 Final model characteristics chart

Results

Model Evaluation and Validation

During model development, a test set was used for evaluation.

The final architecture and hyperparameters were chosen because they performed the best among the tried combinations. For a complete overview of the final model architecture and parameters, refer to the file “model_deploy.prototxt”.

Another aspect of evaluation is the analysis of confusion matrix and misclassified examples. From the visualization below it is seen, that the model can get confused on extremely unclear or blurry

images or when other objects like glasses or hair cover the eye. Also, the model has a small classification bias – it tends to predict a closed eye if the picture is unclear (164 > 111).

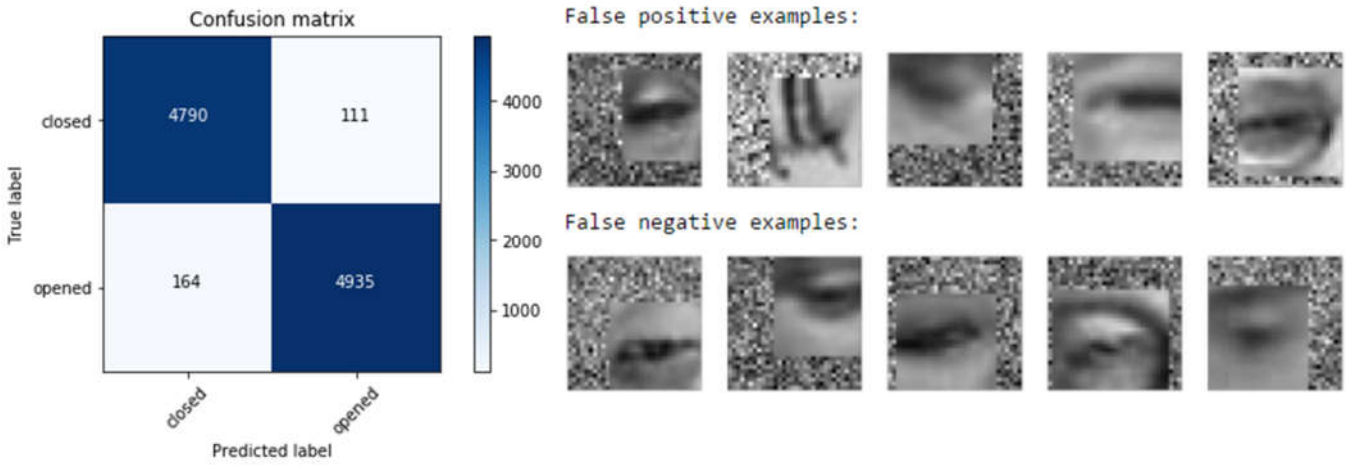


Fig. 9 Confusion matrix with misclassified examples. Notably, for a human, it is also quite hard to tell whether eye is open or not.

To validate the model and application robustness, a test was conducted with 3 different users (one is wearing glasses) in different light conditions. Mouse, touchpad or keyboard was not used during the test. After a 5-minute training, the test cases were:

1. Open an internet browser, read a short article from a web page on speed dial.
2. Open file manager and copy a file from one location to another.

The results can be summarized as follows:

- All users managed to finish the test within the reasonable[8] time.
- User with glasses experienced few eye state classification errors; error rate depends on light source position and glasses shape/size.
- Model predictions are stable within the wide range of illumination intensity (daylight, bright artificial light, work lamp, night lamp), however, user's face should be lit evenly and the light source should not blind the camera.
- Model is sensitive to camera frame rate. Low FPS results in blurred and noisy images.

In order to get the test results representative and reproducible, some of the actual classified images were saved for each user. Few images are presented in the free-form visualization section (b). All of the saved images available in "dataset/validation/" folder.

Considering the analysis above it is concluded that the model is robust enough for the stated problem.

[8] The total time spent on mouse action attempts (pointing, clicking, scrolling) was less than 20 seconds.

The results found from the model can be trusted, as its train and test data was real and sampled in different conditions (the dataset is called "Closed eyes in the wild"). Also, the successful test was conducted with a few actual users. This test covered main use cases for the model.

The final model appears to be reasonable and in line with the solution expectations. This conclusion is made after comparing the model accuracy (97.2%) with the previously defined expected value of 95% and the analysis of misclassified examples.

Justification

On my laptop (CPU: Intel(R) Core(TM) i7-6700HQ @ 2.60GHz, OS: Windows 10) with built-in VGA (resolution 640x480) web camera I got the following results:

- Classification delay for both eyes is around 5 milliseconds.
- Overall processing delay for one frame is 11 milliseconds in average.
- In usual office light conditions, the classification accuracy is above 95%[9].
- Optimal distance from the camera to the user's face is within 50-100 cm.
- Supported framerate range is 10-30 FPS. Most comfort operation starts from 20 FPS.

In summary, the application does solve the stated problem and outperforms the defined benchmark results. It provides a viable user input alternative and extends a variety of human-computer interfaces. However, the built implementation is rather a proof-of-concept and has a lot of improvement potential (see the Improvements section).

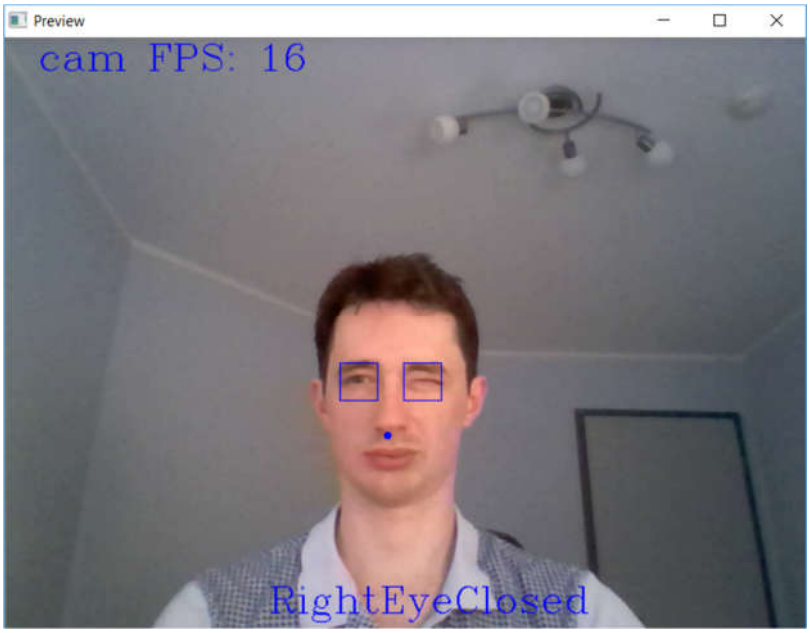
Conclusion

Free-Form Visualization

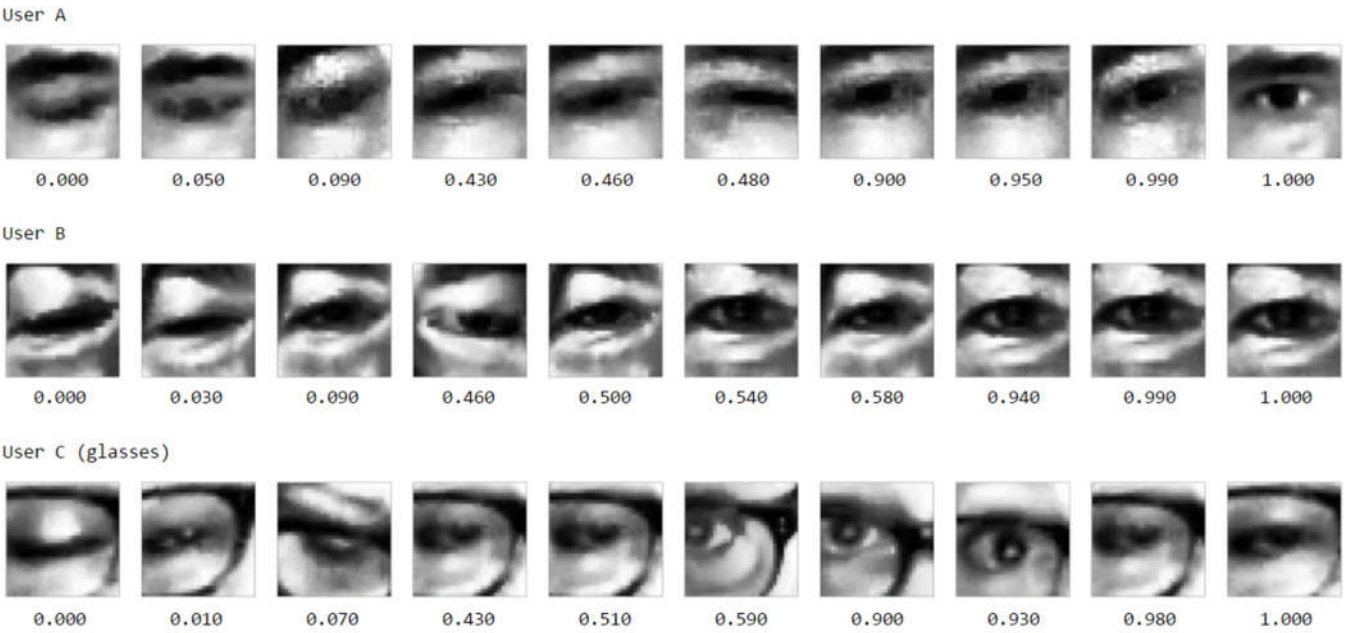
Below is presented the application preview window (a) with the blue squares of extracted eye regions. The text shows current camera frame rate and detected eye state.

Visualization (b) shows examples of real classified images from the camera that were saved for each user during the test. Classification score is reported under each image.

[9] This value is an indicative estimate as it varies due to dynamic factors and application use cases. For a future research, videos of the application preview window could be recorded and analyzed later to get more precise accuracy estimate.



(a)



(b)

Reflection

The project development can be summarized as follows:

1. The stated problem relevant dataset was found.
2. General application architecture and data flow were designed.
3. Appropriate frameworks and toolsets were explored and selected (Caffe, OpenCV, Dlib…).

4. The data was downloaded and preprocessed.
5. A benchmark model was created for the classifier
6. An optimized classifier model was trained by adjusting benchmark model parameters.
7. The application was implemented according the initial design.

The most challenging task in this project was the actual application implementation. I had to solve many satellite tasks to get the classifier and app to work reliably (face tracking, landmark detection, sub-pixel point tracking, filtering…)

Steps 3 and 6 I found the most interesting, as I got a good opportunity to explore great tools and existing state-of-the-art algorithms in image processing and apply them to solve the real, concrete problem.

Improvement

As this project implementation is rather a proof-of-concept, a lot can be done to improve overall experience. Only the main improvement directions summarized as follows:

- Dataset for model training. It was shown, that available dataset contains sometimes too low quality images. They just not very similar to what actual classifier will classify. There should be more examples with glasses.
- Adaptive model – it would be great if the application could adapt to the exact hardware or user by learning the data in his conditions.
- Application UI – there should be a nice user interface, which allows configuring all the settings, allow presets, different profiles (surfing, gaming… etc). This is a must-have for a real user-friendly application.
- Performance. By optimizing the algorithm and moving the implementation from Python to C++ or other lower level language, it should be possible to reduce the processing delay greatly.