Automatic Pain Assessment in Faces Using Computer Vision and Deep Learning

TIENPRAU01 - Q3 2015

Project Report

Author:
Kasper Nielsen | 10731

Supervisor:
Henrik Pedersen

Aarhus University - Department of Engineering

April 7, 2015

Contents

1 Introduction

1.1 Purpose and Goal

The purpose of this R&D project is to examine the possibility of creating a system that automatically and objectively assesses pain in humans by observing their faces in pictures or short video sequences.

There are several goals for this project. These are:

1. To examine the literature on the state of the art of pain assessment with vision systems in order to assess which methods and features seem promising.
2. Gain insight into the theory of machine learning with deep artificial neural networks or *Deep Learning*.
3. Attempt to implement a system applying deep learning to the problem of automatic visual pain assessment.
4. Comparing the above mentioned system to the state of the art using a publicly available database of images for pain assessment.

1.2 Scope

This project will be carried out as an *Engineering Research and Development Project (TIEN-PRAU01)* and as such has a scope of *5 ECTS* points over *one school quarter*.

This means that the scope of the study into the state of the art of systems for pain assessment with computer vision will be limited to reviewing just a few essential papers.

Further, a system for pain assessment with computer vision using deep learning will only be attempted implemented in MATLAB as a proof of concept.

2 Theory and Related Works

Humans are very good at discerning emotions in others from looking at pictures or sequences of pictures of faces, among these, the sensation of pain. This tells us that there must be visual cues in these, that it is possible to detect and use for classification.

To find a way of detecting at database of images of people displaying real pain along with reliable labels is needed. One such database is the UNBC-McMaster database described in Section 2.1 below.

2.1 The UNBC-McMaster Shoulder Pain Expression Archive Database

The UNBC-McMaster Shoulder Pain Expression Archive Database [16] is database of image sequences of subject being observed experiencing pain. There are 25 subjects, all suffering from shoulder (rotator cuff) injuries in one shoulder. The subjects are asked to move both arms, either manually or by manipulation by a physiotherapist. Both arms are moved to provide both positive and negative examples of pain.

Figure 1: Example frames from the database

For each image sequence ie. each movement of an arm, the subjects are asked to report, on a scale of $0 - 10$, their perceived level of pain. An objective observer also evaluates the level of pain the subjects experience. These are used to control for subjective variances in perception and reporting of pain. Both data are available in the database

From the 25 subjects there are in total 200 sequences, with some subjects having more sequences than others. In total there are 48398 frames. For each frame facial landmarks are tracked by manually fitting a mask of landmarks to the first frame and then tracking the points throughout the sequence.

Also for each frame, Action Units (AUs) of the Facial Action Coding System (FACS) [6] shown to be related to pain (more on this in Section 2.1.1 below), are evaluated and are available in the database. The FACS AUs each signify a movement or an action of the face eg. the closing of an eye, the raising of an eyebrow or the opening of the mouth. The AUs, if present, are all evaluated on a scale of A to E (or somtimes more practically $1-5$) with A meaning *trace* and E meaning *maximum*.

For a thorough analysis of the database the reader is referred to [17].

2.1.1 Pain Score

The easiest way of labeling the frames in the UNBC database would be to just label all frames in a sequence with the value i subject reports. But there are several problems with this method. Firstly there is the problem of individuals experiencing and reporting pain differently, because there are no common reference. This can be mediated by using the report of an objective observer, with training in evaluating pain.

The real problem with the method above is that they assume an equal amount of pain through a whole sequence. In reality the level of experienced pain will very throughout the sequence with a peak at some point. The level of this peak is then likely what subjects report. In order to fully utilize all the image data in the database an indication of pain for each frame is needed.

Prkachin [20] have show that there is a high correlation between certain FACS AUs and the level of experienced pain. They propose Prkachin and Solomon Pain Intensity (PSPI) score (1) as a tool for evaluating pain in subject from their facial expression.

$$PSPI_{score} = AU4 + \max(AU6,\ AU7) + \max(AU9,\ AU10) + AU43 \tag{1}$$

The Action Units account for brow lowering (AU4), orbital tightening (AU6 and AU7),levator contraction (AU9 and AU10) and eye closure (AU43). The values are numerical (0 = absent, 5 = maximum). The value for eye closure (AU43) is binary (0 = open, 1 = closed). This gives the score a scale of $0-16$.

Because FACS codes are available for each frame, the PSPI score is a good choice for labeling data at the frame level. This is further supported by that fact that we know that the emotions felt in the experiment is pain. Therefore we can with high probability attest the AUs to the pain and not other factors/emotions.

2.1.2 Problems With Using the UNBC Database

The UNBC database is a good and important tool for the development of pain recognition systems. It does however leave one wanting at some points.

Firstly, of the 48398 frames in the database 40029 are frames with no pain at all. And of the ones showing pain most are very low levels (see Figure 2). This results is a high a priori probability of no pain in the data.

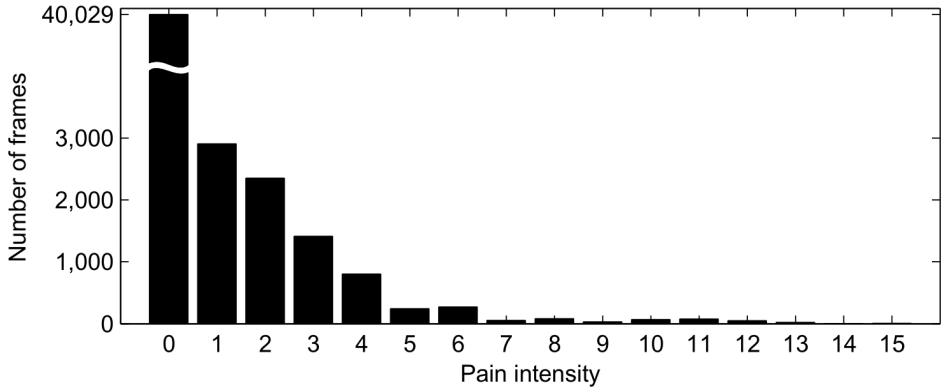

Figure 2: Histogram of PSPI scores in UBNC database. Figure taken from [9]

Secondly, the database only show subject in pain. This means that without negative examples of other than neutral faces, there is a risk of over-fitting certain features that may be more generally show for a wider range of emotions, leading to false positives. A solutions to this could be as in [7], where the authors use the UNBC database to learn specific pain related features, but also the more general *Cohn-Kanade AU-Coded Expression Database* [10] to ensure better generalization of their model. Such an approach is however beyond the scope of this report.

Finally, there is a lot of frames in the database in which the subjects turn or bow their heads enough to interfere with the processing and classification. This is also documented in [17]. The problem is that this was not monitored/recorded in the creation of the database, so it hard to automatically adjust for or ignore. Estimating head pose and correcting for this like in [2] is beyond the scope of this report.

2.2 Related Works

In [16] the creators of the UNBC database use the registered landmarks in the images to make procrustes aligned representations of the faces along with facial appearance, shape normalized using Active Shape Models (ASM) and Active Appearance Models (AAM), as feature vectors. These feature vectors are then used to train a Support Vector Machine (SVM). With this they achieved detection rates of 57.1% to 87.5% for individual FACS AUs and up to 83.9% for binary pain detection.

In this report I attempt to reproduce these results using the same data and with a similar approach but with my own implementation.

Kaltwang et al. [9] similarly use the UNBC database. They compare the performance of using shape based features (PTS), taken from the registered landmarks, with appearance features. The appearance feature were both 2D Discrete Cosine Transform (DCT) and Local Binary Patterns (LBP). The authors employ Relevance Vector Regression (RVR) to estimate pain for type of feature (shape, DCT and LBP). A fusion of the individual features pain estimates is then made, again using RVR. The authors conclude that appearance based features outperform shape based. They do however believe that improved registration of shape, that compensates for out of plane motion (eg. [21]), would improve performance. Regarding the appearance based feature, the authors state that in most cases the LBP features worked better than DCT features, taken individually.

A major shortfall of the methods described above is that they all rely on the facial landmarks that are registered with human intervention. Without self implementing a strong and reliable way of automatically registering these, systems like the ones described above are not applicable for use in real-life, real-time applications. Therefore I try a completely different approach in this report.

An inspiration for this approach is [22]. In this, the authors predict human eye fixation using deep Convolutional Neural Networks to extract high level features. This approach does not use human selected features, but instead uses human eye tracking data to learn the high level features that humans fixate on. A similar approach could possibly be tuned to register the same cues that humans use to recognize pain in faces.

2.3 Proposed approach

The novel approach in this project is to use only the image data from the UNBC database as input data and only use the metadata for ground truth labeling of the training, test and validation data.

The method can be summarized as follows:

1. A Viola-Jones detector [23] will be used to locate the face in the images.
2. A patch around the face will be extracted.
3. The extracted patch will be resized to a fixed size eg. 100×100 or 48×48 pixels.
4. The resized patches will be used to train a Convolutional Neural Network (CNN)

Several ideas for the scope of each classification network exist. One idea is to train a large network with all the data on all types of subject, to have a very generalized model.

Another is to train a couple of networks one a few stereotypes of subjects ie. males, females, children and perhaps also different age segments separately. This would be less general, but maybe better tuned to the idiosyncrasies of each group.

A likely use-case for a pain-recognition system, like the one proposed, is for monitoring patients suffering from paralysis or mental handicap, as they may have difficulty in reporting and communicating pain levels by themselves. Additionally, these patients are known to exhibit atypical responses to pain. A final idea is therefore to train a model on a per subject basis. This would likely not generalize very well to all other subjects. Instead it may be better equipped to handle the sometimes very individual response of subjects to certain levels of pain. For this to work though, it would require that it is possible to train the network with few enough data, that it would be feasible to acquire for each subject. A way of achieving this would be to, instead of training a network from scratch, refine a more general pre-trained network with samples from the subject in question.

An example of this kind of fine-tuning of a general network is Karayev et al.'s [11] adaptation of a network, trained to classify objects in the ImageNet database [5], to instead recognize the *style* of an image.

3 Normalized Appearance and Shape Method Using SVM

In this section I describe how I tried to reproduce the results of [16] by implementing my own algorithm for extracting data and training a linear Support Vector Machine (SVM) using LIBSVM [3].

The data extracted from the database and used in the experiments are normalized shape and shape normalized appearance (described in greater detail in Section 3.1 below). Two experiments will be done with these data. One will focus on detecting the presence of FACS Action Units relevant to PSPI model described in Section 2.1.1. The other will focus only on the detection of pain. Both experiments will be done using either the shape data, the appearance data or using shape and appearance data combined.

3.1 Data pre-processing and Feature Extraction

3.1.1 Shape Data

The raw shape data consists of `.txt`-file for each frame with two columns of (x,y) pixel coordinates, 66 points in total each for a specific facial landmark (See Figure 3). These combined make a mask of the facial shape. They are however centered in the corner of the image and shifted, scaled and rotated arbitrarily form subject to subject and from frame to frame because of movement.

3.1.1.1 Procrustes Alignment

In order to use the shape data they need to be shifted to a zero mean reference and be normalized with regard to size and rotation, without affecting shape. For this, Procrustes alignment is just what is needed. Procrustes alignments works by aligning sets of points to a common reference (either given, or taken as the mean of the set to be aligned) by a rigid transformation ie. translation, rotation and scale.

In my project I do this iteratively and by using MATLABs build in `procrustes` function, like so:

1. For each shape $\mathbf{S}_i$ subtract the mean of all points in $\mathbf{S}_i$.
2. Calculate mean shape $\mathbf{Z}_0$, by taking the mean over all shapes $\mathbf{S}_i$ for each point $\mathbf{S}_{i,j}$.
3. Align all shapes $\mathbf{S}_i$ to $\mathbf{Z}_0$ using `procrustes` to get the aligned shapes $\mathbf{Z}_i$.
4. Calculate the new mean shape $\mathbf{Z}_0$ over all shapes $\mathbf{Z}_i$ for each point $\mathbf{Z}_{i,j}$.
5. Repeat from step 3 for as many times as is needed. A typical stopping criteria is when $\mathbf{Z}_0$ stops changing significantly.

Two iterations will typically suffice and is therefore what is used in this project.

The result is what [16] and [1] call the *similarity normalized shape* or **S-PTS**. This is a vector of x and y-coordinates $\mathbf{s}_n$.

3.1.2 Appearance Data

In order to be able to compare and classify the appearance of different subjects with an SVM, the image data from each frame has to be normalized. The representation of image data need to be invariant to several factors:

- The subject's face can be in different parts of the image frame.
- Subjects' faces can vary in size, both from physical size and from proximity to the camera.
- The subject's face can be rotated/tilted sideways as a result of posture and moving the arm.
- The background is not uniform and varies from sequence to sequence.
- The general shape of a subjects face varies from subject to subject.

The goal is to sample to image at a set amount of points that maps to a fixed pixel mask, called *objectPixels*. This way the appearance is represented as a vector of pixel intensities, that maps to a common face shape with the background excluded.

Two strategies exist for doing this. One [1] calls *similarity normalized appearance* or **S-APP**. Here the pixels are sampled inside the shape bounded by the facial landmarks and transformed rigidly according to the Procrustes alignment described in Section 3.1.1.1 above. This method seems to preserve most variation caused by facial actions but also results in more spatial variation of the position of features in the image representation, called $\mathbf{a}_n$.

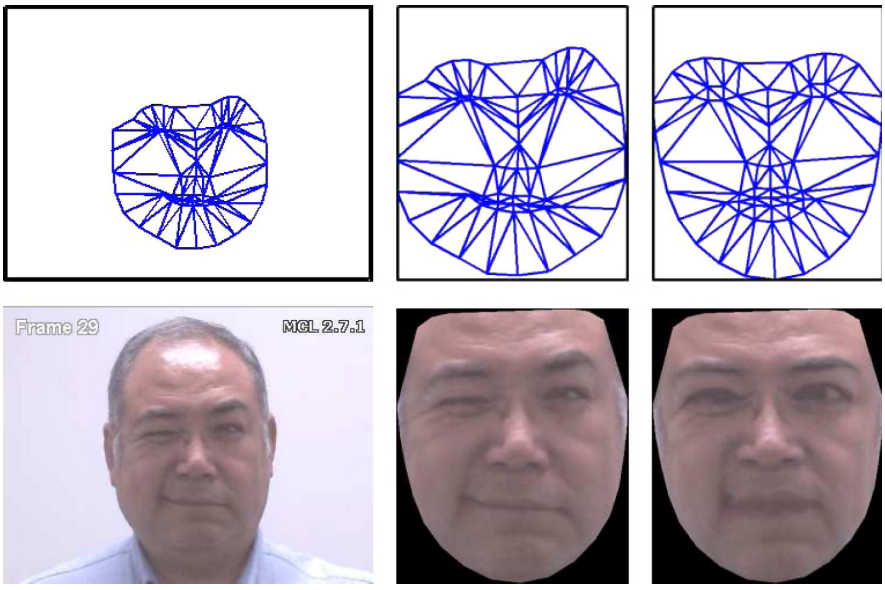

Figure 3: Example of image representations. Top row: Left, triangulation of raw landmarks. Middle, Procrustes aligned shape. Right, mean shape. Bottom row: Left, raw image. Middle, **S-APP**. Right, **C-APP**. Figure taken from [1].

Another strategy what [16] and [1] call the *canonical appearance* or **C-APP**. This method samples the intensities inside the shape bounded by the facial landmarks and warps their position in a non-rigid fashion, to match the mean shape $\mathbf{Z}_0$. This method removes more of the spatial variation of the position of features in the resulting image representation, $\mathbf{a}_0$

A comparison of the two strategies is shown in Figure 3. As [1] rightly notes, it would seem that **C-APP** removes too much variation, eg. in the case of an eye closure as shown in Figure 3. The case is though that **C-APP** preserves the most important part of this action, because the wrinkles around the eye is still very clear. These serve to distinguish between a *twitch* as a result of pain and a regular blink, which does not result in these wrinkles. This distinction is a common cause for false positives in a pain recognition system. Further, the supposed lost information is conserved if the shape data and appearance is used together.

In this project the **C-APP** representation is used.

3.1.2.1 Implementing Image Warping

The non-rigid warp of the facial appearance to the mean shape is done with at piecewise affine transformation. This means segmenting the face into triangles and the for each triangle warping the pixels inside to the corresponding triangle in the reference mean shape. The procedure is implemented as follows: [1]

1. The landmark points of the mean face shape needs to be put into a graph that connects all vertices into triangles without overlapping edges. There exists many solutions for this problem, but by making a Delauney triangulation, one can find the single solution that maximizes the smallest angle in all the triangles. This solution tends to avoid *skinny* triangles and is a good fit for this application.

   The Delauney triangulation is made using MATALB's `delaunayTriangulation` function. It takes a set of points as input and returns a *triangulation* object, which contain information about which vertices are connected in triangles (also called a *Connectivity List*), and methods to determine in which triangle a certain point is inside.

2. *objectPixels* is determined by evaluating which pixel position is not inside any triangle of the Delauney triangulation of the mean shape. This results in a matrix of xy-coordinates, one row for each pixel.

3. For each frame:

   I. The image is loaded into MATLAB, converted to gray scale to conserve memory and converted to double-precision form int8 to facilitate calculation.

   II. A new *triangulation* object is then created using the connectivity list of the Delauney triangulation of the mean shape and the facial landmark points. This new triangulation is not an optimal Delauney triangulation. Instead it is a triangulation of the point in the frame where the triangle connections correspond to that of the Delauney triangulation of the mean shape.

[1] This is a general description of the implemented method. For full MATLAB implementation see files `WarpImagesToMeanBatch.m` and `warpAppToMeanShape.m` in Appendix A

III. For each triangle in the Delauney triangulation of the mean shape:

i. An affine transformation matrix $\mathbf{A}$, from the triangle to the correspond triangle in the new triangulation is calculated.

$$\mathbf{A} = \begin{bmatrix} x_{a1} & x_{b1} & x_{c1} \\ y_{a1} & y_{b1} & y_{c1} \\ 1 & 1 & 1 \end{bmatrix} \Big/ \begin{bmatrix} x_{a2} & x_{b2} & x_{c2} \\ y_{a2} & y_{b2} & y_{c2} \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} * & * & 0 \\ * & * & 0 \\ * & * & 1 \end{bmatrix} \tag{2}$$

In (2) x_{a1}, x_{b1} and x_{c1} is understood to be the x-coordinates of point a, b and c, respectively, in the reference triangle. Likewise y_{a2}, y_{b2} and y_{c2} is the y-coordinates of point a, b and c, respectively, in the corresponding triangle.

ii. The transformation matrix $\mathbf{A}$ is then used to create a MATLAB *transformationObject* using the `affine2d` function. This object has methods for handling the mapping of pixel positions with the affine warp.

IV. Then for each pixel position inside the new triangulation, the new pixel position is calculated using the appropriate *transformationObject* and is stored along with its intensity.

V. An interpolation has to be made to map the decimal coordinate value of the warped pixels to the integer grid of an image. This is done by making a MATLAB *scatterdInterpolant* object. This takes all the decimal value positions and intensities of the warped pixel. You then define with interpolation method it is to use. In the project a *linear* interpolation is used.

VI. The *scatterdInterpolant* object can then be evaluated at all the coordinates of *objectPixels* to output a vector of gray scale pixel intensities.

4. It can happen that an error occurs when trying to calculate the transformation matrix $\mathbf{A}$. This is typically due to excessive rotation/tilting of the subjects head, causing the matrix division to be impossible. In this case the error is simply logged and the script moves on to the next frame. The number of frames for which errors occur are around a couple hundred out of over 48000 frames. They are therefore deemed negligible and are omitted in further training and test.

3.1.3 Combining Data

With the two datasets described in Sections 3.1.1 and 3.1.2 above, it is possible to make, in total, three datasets, excluding the case of no data. That is:

1. **S-PTS**: Only the Procrustes aligned shape data. This data is a vector $\mathbf{s}_n$ stacking the 66 x-coordinates atop the 66 y-coordinates, resulting in a 132 dimensional feature space.

2. **C-APP**: Only the shape normalized *canonical* appearance. This data is a vector $\mathbf{a}_0$ of 11646 dimensions that represent the pixel intensities at the coordinates listed in *objectPixels*.

3. **S-PTS + C-APP**: A combination of the shape and appearance data. The appearance data vector is stacked on top of the shape data vector to form a 11778 dimensional feature vector

$$\mathbf{x}_{SPTS+CAPP} = [\mathbf{a}_0 \ \mathbf{s}_n] \in \mathbb{R}^{11778} \tag{3}$$

These datasets will then be used in the experiments of Sections 3.2 and 3.3 below.

3.2 Experiment 1: Recognizing FACS Action Units

3.2.1 Purpose and Method

The purpose of this experiment is to see how well a linear SVM can be trained to recognize FACS Action Units from the data described in Section 3.1. The way this will be tested is by training a SVM using data from most of the subjects, and then validating against data from one or more other subjects that are not in the training set. The plan was initially to train a binary SVM for each FACS AU and do leave-one-subject-out cross validation on all subjects. This, however, was not possible for the following reasons

1. No one subject presents all the FACS Action Units relevant to the PSPI-score. Therefore validating all the classifiers against one control subject is not possible.
2. The time is takes to train the 10 SVMs varies between 2 to 6 hours, depending on which dataset is used. Therefore it was not possible for me, within the time frame I have available, to do exhaustive cross-validating.

The solution to this was to do the following: [2]

- Select two subjects to be used as controls for validation, so that examples of all FACS AUs are present in both the training and validation set. I chose subjects 1 and 8 to be in the validation set. The rest are in the training set.
- For each FACS AU a set of training data was formed by taking all the positive examples in the training set and an equal number of randomly selected negative examples from training set. This ensured a uniform prior probability, and reduced the amount of data that was used, improving training time.
- The classifiers where trained using LIBSVM [3] in MATLAB. The implementation did not utilize the 6-core CPU very well, so the training was done in parallel using MATLABs `parfor` loops, with each SVM training on a separate thread, 6 at a time.
- The classifiers were evaluated by calculating the Receiver Operating Characteristics (ROC) Area Under the Curve (AUC). This relates the rate of True Positives (TP) to the rate of False Positives (FP) over a range of sensitivities, giving an easily comparable result between 0 and 1.

[2] See full MATLAB implementation in Appendix A: `compareSVMscore.m`

3.2.2 Results

3.2.2.1 Recognizing FACS AUs with S-PTS data

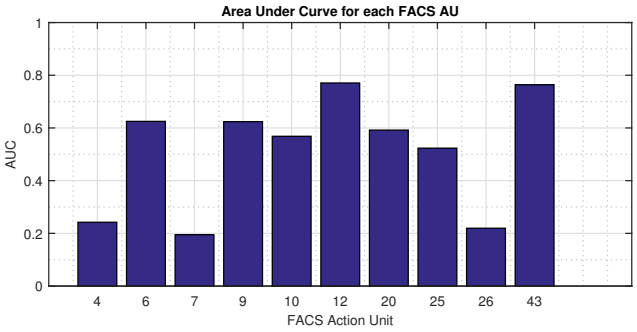

Figure 4: AUC for **S-PTS** tested on subjects 1 and 8. SVM Trained on remaining subjects

3.2.2.2 Recognizing FACS AUs with C-APP data

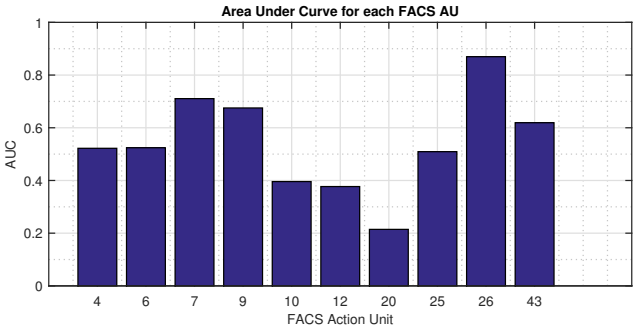

Figure 5: AUC for **C-APP** tested on subjects 1 and 8. SVM Trained on remaining subjects

3.2.2.3 Recognizing FACS AUs with S-PTS + C-APP data

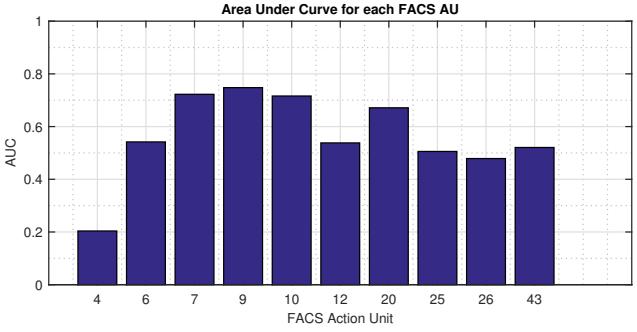

Figure 6: AUC for **S-PTS + C-APP** tested on subjects 1 and 8. SVM Trained on remaining subjects

3.3 Experiment 2: Detecting Pain in faces

3.3.1 Purpose and Method

This experiment is very similar to Experiment 1 (See Section 3.4.1), in both purpose and method. Again a SVM is trained to classify based on the datasets described in Section 3.1. The difference here is that in stead of training to detect individual FACS AUs, the SVMs are trained to detect pain, in the form of the PSPI-score. [3]

3.3.2 Results

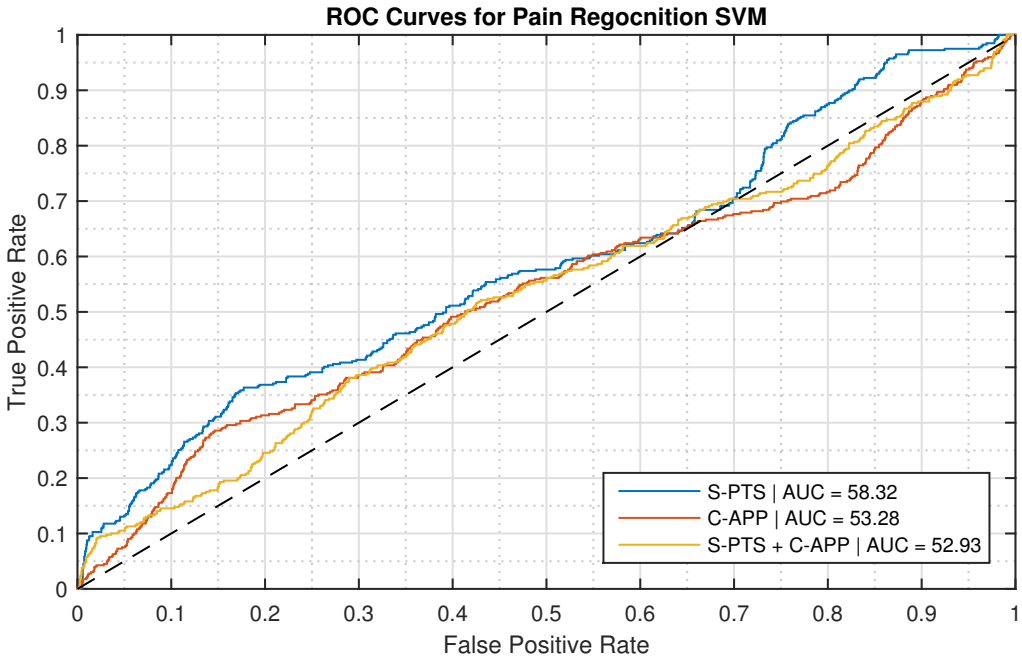

Figure 7: Receiver Operating Characteristics curve for the SVMs trained on each data set. Area Under the Curve (AUC) given in percent.

3.4 Discussion

3.4.1 Experiment 1

Comparing the results of this experiment to those of [16], I have not been able to reach the same levels of performance. They see AUCs above random (50%) for all AUs and all datasets. I only see a significant increase over random in for some AU with some datasets, and for some they do not even achieve that.

[3] See full MATLAB implementation in Appendix A: `compareSVMscorePain.m`

There are however some similarities. Like [16] I find that the different data representations have complementary strengths and weaknesses, meaning that some Action Units are better described by shape variation, and some are better described by variation in appearance. I also see that overall performance evens out for the combined dataset, and that AUs that are strongly detected with either shape or appearance data do not necessarily have a strong response with the combined dataset.

There may be several reasons for why my results differ so much from those of [16]. In my experiments the data is converted to gray scale. This is done to reduce the complexity of implementation and the amount of data that needs to be processed and hence processing time. It would seem that they used to full color images, although it does not say explicitly in the article. It may very well be that to much information is lost in converting to gray scale, and that this adversely affects performance.

As the authors also note, there is a good portion of the samples where out of plane motion is significant. This seem to have introduced some artifacts in the image data when warping them the mean shape (See Figure 8).

Figure 8: Examples of faces with artifacs stemming form out of plane motion and the subject wearing glasses.

In the scope of this project it has not been possible for me to deal with this, which means that these will likely have interfered with the performance of the classifier, although in what way and to what extent I cannot say say with certainty. A test could be to manually exclude sequences where the subject moves excessively.

In general there is a problem of availability of data. As noted in Section 2.1.2, although the amount samples is large (> 48000) the amount of positive samples is comparatively low (< 8500), most of which are of low intensity. This means that amount of available training examples compared to the dimensionality of the data makes it hard to train a good model, especially for the very high dimensional **C-APP** and **S-PTS + C-APP** datasets.

3.4.2 Experiment 2

The results of this experiment show that with this implementation it is not possible to detect pain with a certainty significantly greater than random. This is disappointing since [16] and subsequent authors have seen promising results with this approach.

I believe that many of the same factors as those described in Section 3.4.1 above are to blame for these results. That being said, I cannot say for sure that there are no errors in my implementation, as there are simply no more time in this project for further experimentation.

4 Convolutional Neural Network Method

In this section I describe how I tried to implement a system for detecting pain in faces using a Convolutional Neural Network. I did not succeed in creating a working proof-of-concept in the time I had available for this project, but I will describe what I tried and what I learned below.

4.1 CNN Basics

Convolutional Neural Networks are a class of deep Artificial Neural Networks (ANN). A traditional feed-forward ANN consists of layers neurons and biases. These are fully connected by a set of weights that are trained by backpropagation (See Figure 9).

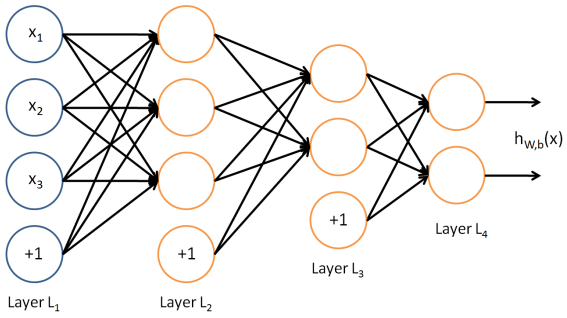

Figure 9: Illustration of a simple ANN, taken from [18].

In CNNs the weights are instead sets of filter kernels that are convolved with the input image, hence the name. This reduces the amount of parameters to train because the weights are shared for all the pixels in an input.

Between the filter neurons there are typically pooling layers inserted. These serve to further reduce and concentrate the data in the network. They work by nonlinearly sub-sampling the output image of a filter layer by eg. only sampling the pixel of highest intensity in each 2×2 pixel patch of the image (See Figure 10). This also makes them more tolerant to small translations of the input image.

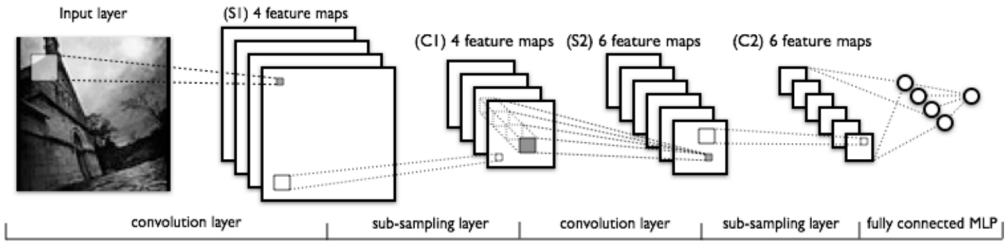

Figure 10: Illustration of a simple CNN, taken from [15].

The final classification is then done by either an SVM or one or more layers of traditional fully connected ANN layers like shown in Figure 10.

4.2 CNN Libraries/Toolboxes

I have tried a couple of different libraries and toolboxes to implement the CNN:

4.2.1 DeepLearnToolbox

DeepLearnToolbox [19] is a toolbox for MATLAB, completely written in MATLAB by Rasmus Berg Palm, as part of his masters thesis at DTU. This toolbox offers a wide variety of functions within the domain of Deep Learning.

This toolbox was my first attempt at implementing a CNN for pain recognition. I was able to install the toolbox and run test and example scripts without issue, but I never got any good results for my application. The problem was that with my data, processing was very slow. It took several hours to train a single epoch, making it next to impossible to make any progress in my development. I attest these problems to the fact that everything ran in MATLAB, which is known to be inefficient at times, and also partly to my inexperience with designing and training CNNs.

4.2.2 Deep Learning for Saliency

As described in Section 2.2, a system for predicting eye fixations using CNNs has been implemented. I tried briefly to use thier code [22], but found quickly that it written very much was written with their application in mind, and did not easily generalize.

4.2.3 MatConvNet

MatConvNet [14] is an Open-Source MATLAB toolbox for CNNs written by *vlfeat.org*, known for their excellent Computer Vision toolbox for MATLAB. The toolbox is implemented efficiently in C++ and in CUDA for GPU acceleration on NVIDIA hardware.[4] The code is compiled locally and have `.mex` wrappers so that it can be executed from MATLAB like any other MATLAB function. This is the toolbox I the majority of my time using.

4.2.4 Caffe

Caffe [8] is much like MatConvNet a framework for implementing CNNs implemented in C++ and CUDA with tie-ins to MATLAB. Caffe seems to be the best freely available CNN framework at the moment, but an official port for Windows is not available yet, so using it was not an option for this project.

[4] See Appendix B for more on how to enable GPU acceleration with MatConvNet

4.3 Method

The overall idea for my system is described in Section 2.3. After initial tests with different toolboxes I chose to work with the MatConvNet toolbox from VLfeat for the CNN part of the implementation.

4.3.1 Viola-Jones Face Detector

A Viola-Jones face detector was implemented in MATLAB using a *CascadeObjectDetector*.[5] This was used to locate the face of the subjects in all the frames. For a few of the frames the detector would have detections aside from the actual face eg. small patches around the neck or in the background. Also, sometimes it would not detect a face if subject was wearing glasses. To help with this, a minimum size for detection was set to 100×100 pixels. This is easily enough to detect a the actual face, but reject most of the other erroneous detections. After this only a very small amount of frames had erroneous detection. Therefore is was elected to just drop the frames that did not have exactly one detection, in stead of manually going through all the data.

The location of the face is marked by a simple bounding box of coordinates. The patch within this box is extracted, converted to gray scale and resized to a common size. Both sizes 48×48 pixels and 100×100 pixels were tried.

The position, rotation and scale of the face inside this patch could vary. Also the background is not extracted. This is because the plan is to train a robust CNN that adjusts for this itself, making the need for pre-processing in a final application minimal.

4.3.2 MatConvNet Data Format

In order to use the MatConvNet toolbox the data has to be formatted in a certain way.[6] Specifically the data has to be arranged in a layered `struct` called `imdb` (see Figure 11). This contains all the image data, the target labels, information on which samples are to be used for training/test/validation, along with some metadata.

The images has to be arranged in a 4D-matrix. The first two dimensions are the dimensions of the images (Rows/Columns). The next is a singleton dimension. This dimension is used when the input fans out into several feature maps. The last is the numbers of frames (N).

$$\texttt{imdb.images.data} \in \mathbb{R}^{R \times C \times 1 \times N} \tag{4}$$

All image data values are stored as *single*-precision (32-bit) variables. This is typical for CNN data as it greatly increases computational speeds, especially with GPU acceleration. The mean image over all frames is also given as input data.

`imdb.images.labels` is a vector of values, either 1 or 2 corresponding to the classes in `imdb.meta.classes` indicating if a frame is a positive or negative example. Likewise `imdb.images.set` is a vector of values, either 1, 2 or 3 corresponding to the classes in `imdb.meta.set` indicating if a frame is to be used for training, for validation or for test.

[5] See Appendix A - `ViolaJonesPatchExtract.m` for full MATLAB implementation.
[6] See Appendix A - `leaveOneOutData.m` for full MATLAB implementation.

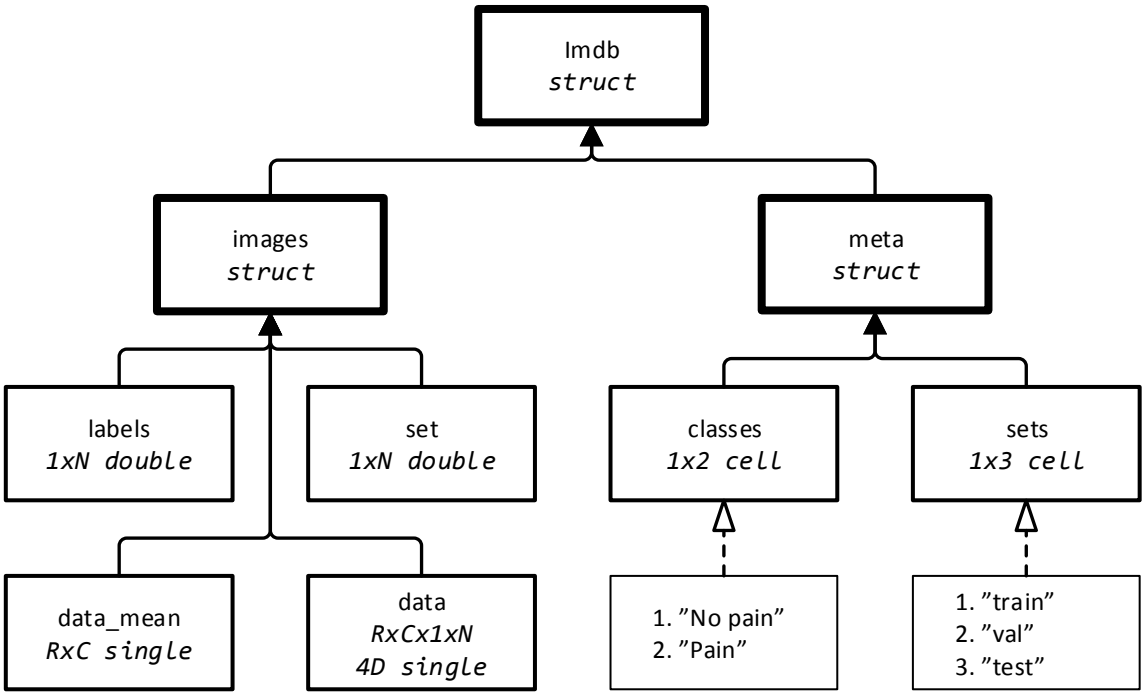

Figure 11: Diagram of image database structure and contents

In my case I only use a training and a validation set, and no separate test set. This does not cause any problems. The feature is just there so you can do cross validation on different sets of validation data and still save a separate test set.

4.3.3 CNN setup

Being a novice in working with deep learning and CNN I chose to base my implementation on a working example that comes with the toolbox and adapt it to my application.

The example base for my implementation was a network designed by the people behind MatConvNet to be similar to Yann LeCun's *LeNet*. This example was made to classify hand written digits from the well know MNIST database [13].

The architecture of the net is defined by creating a *struct* called `net` with fields corresponding to layers (See Listing 1 or Appendix A - `cnn_mnist3.m` for full script). Each layer is itself a `struct`, that defines its type, dimensions and initialization.

Listing 1: Example of CNN setup

```
f=1/100 ;
net.layers = {} ;
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(12,12,1,10, 'single'), ...
                           'biases', zeros(1, 10, 'single'), ...
                           'stride', 1, ...
                           'pad', 0) ;
net.layers{end+1} = struct('type', 'pool', ...
                           'method', 'max', ...
                           'pool', [2 2], ...
                           'stride', 2, ...
                           'pad', 0) ;
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(8,8,10,25, 'single') ,...
                           'biases', zeros(1,25,'single'), ...
                           'stride', 1, ...
                           'pad', 0) ;
net.layers{end+1} = struct('type', 'pool', ...
                           'method', 'max', ...
                           'pool', [2 2], ...
                           'stride', 2, ...
                           'pad', 0) ;
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(4,4,25,250, 'single') ,...
                           'biases', zeros(1,250,'single'), ...
                           'stride', 1, ...
                           'pad', 0) ;
net.layers{end+1} = struct('type', 'relu') ;
net.layers{end+1} = struct('type', 'conv', ...
                           'filters', f*randn(1,1,250,2, 'single') ,...
                           'biases', zeros(1,2,'single'), ...
                           'stride', 1, ...
                           'pad', 0) ;
net.layers{end+1} = struct('type', 'softmaxloss') ;
```

Listing 1 show the net I ended up using, after much tweaking and playing around with parameters. The net starts with a convolutional layer with one input (the image data), 10 output feature maps and a kernel size of 12×12 pixels. Next there is a max-pooling layer with a factor of 2×2. Next there is another convolutional layer with 10 input feature maps, 25 output feature maps and a kernel size of 8×8 pixels. Next, another max-pooling layer like the one before. Next there is another convolutional layer with 25 input feature maps, 250 output feature maps and a kernel size of 4×4 pixels. Next there is a layer Rectified Linear Units, followed by a fully connected convolutional layer. Finally, the last layer is a softmax loss layer. All the convolutional layers are initialized randomly and the biases are set to zero.

There is also a set of hyper parameters to set before training can begin. These are defined in `struct` called `opts.train` as shown in Listing 2.

Listing 2: CNN setup hyper parameters

```
opts.train.batchSize = 100 ;
opts.train.numEpochs = 1000 ;
opts.train.useGpu = false ;
% opts.train.useGpu = true ;
opts.train.learningRate = 0.0001 ;
opts.train.momentum = 0.0;
opts.train.weightDecay = 0.005 ;
```

4.3.4 Experiments

First I just ran the example as is, to verify that the installation was correct and to get a sense of how the code works. After some familiarization with the toolbox I started to adapt it my application. The first thing I changed was the amount output feature maps of last convolutional layer from ten to two, to match my binary Pain/No pain classifier.

I also increased the number of feature maps for all layers. This is because I assume that the task of recognizing pain in images of faces takes a higher capacity than recognizing hand written digits. I kept the basic structure of the net because I do not have much experience with CNNs and opted for a *stick with what you know/know works* strategy.

The two main experiments I tried to conduct was these:

1. Test if I could train a CNN to recognize pain in the face of one subject, by training with the data from all the other subjects.
2. Test if I could train a CNN to recognize pain in the face of a subject, by training on a small sample of frames from that same subject.

4.4 Results

As described at the top of this section I did not succeed in creating a working proof-of-concept. This is due to several factors, that I will discuss in Section 4.5 below.

What I did succeed in was training a network like the one in Listing 1 for a couple hundred epochs. It was difficult to make sense of how the output of the network was to be interpreted, and the classification script from the example did not seem to fit my application. Unfortunately, since the toolbox have only recently been released, there exits little documentation, aside from the code itself and the examples. So I tried to make sense of it myself. For each frame the output consisted of two 15×15 frames, one for each class. I summed the response for each class, for each frame, and looked at which was bigger to classify the frames.

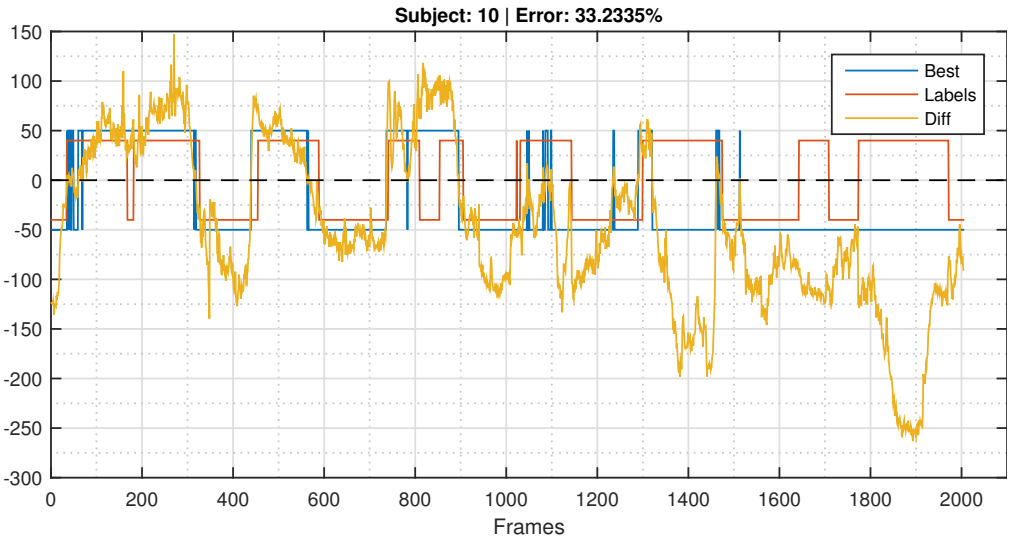

Figure 12: Classification results for for leave one subject out validation with subject 10

Figure 12 shows the classification results for Subject 10. The *Labels* plot shows the target class and the *Best* plot shows the predicted class, based on the method above. Both plots have been scaled for visibility, but it holds for both that a positive value indicates *pain* and a negative value indicates *no pain*. The *Diff* plot shows the difference in response for the two summed feature maps. Again, a positive value shows a preference for the *pain* class and a negative value shows a preference for the *no pain* class.

For this classification the confusion matrix is shown in Figure 13. Here a '1' means *pain* and a '0' mean *no pain*

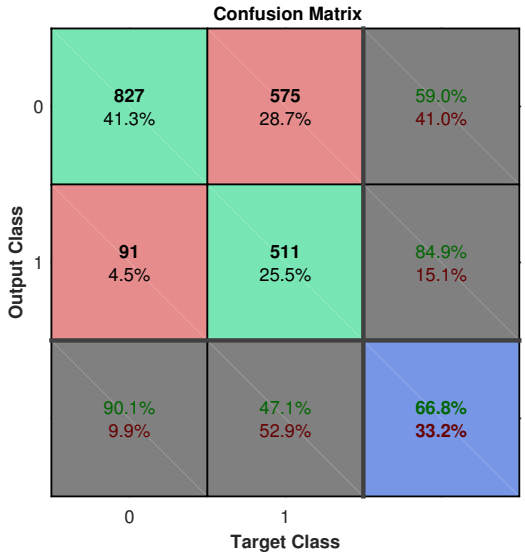

Figure 13: Confusion matrix for validation with Subject 10

The results of Subject 11, who was part of the training set, is also presented in Figures 14 and 15.

Figure 14: Classification results test with subject 11

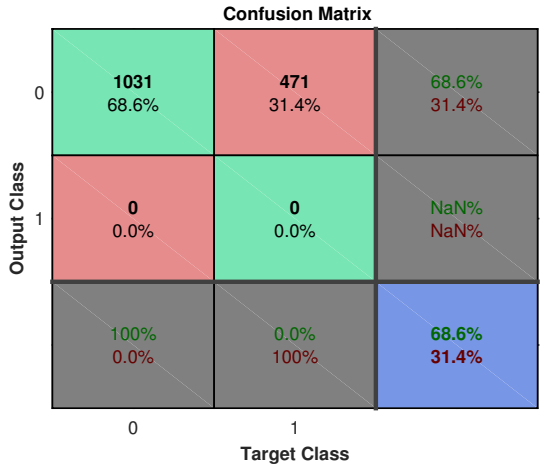

Figure 15: Confusion matrix for test with Subject 11

4.5 Discussion

The results of my work with Convolutional Neural Networks for pain recognition can best be described as inconclusive. There is, however, something to take away from the results.

The classification results from the validation with Subject 10 (Figure 12) show that even though good error rates are not achieved, there seems to be positive correlation between the output of the network and the desired label. This could indicate that with further tweaking/training of the network could perform better. More data may also be necessary to generalize better.

The same seems to be the case when you look at the results for subject 11 (Figure 14). Here it would also seem that, even though the classification fails completely, the network output curve somewhat tracks the target class. Notice especially the peak around frame 250 and the plateaus around frames 500 and 1250. This could again imply that more training with perhaps more data could could make the net perform better.

A factor that would likely also improve the performance of the net is greater control over regularization parameters during training, and perhaps the use of a Convolutional Auto-Encoder (CAE) for initialization. I know from looking at the code examples for larger networks, designed for the ImageNet [5] and Cifar [12] databases, that this is possible, but I have not had time within the scope of this project to figure out how this would work this toolbox. This is left as a task for those whom are to continue the work on this project.

5 Conclusion

In this project I attempted to reproduce the results of [16] and succeeded to some degree. The were however some significant shortcomings, the reasons for which was discussed in Section 3.4.

I also attempted to implement a pain recognition system for facial images using a deep Convolutional Neural Network. I did not succeed in creating a working proof-of-concept within the time available for this project. Still, I was able to present *some* results.

I do believe that I made some valuable progress, towards reaching the goal of implementing a CNN for pain recognition, that others can build upon after me.

All in all the outcome of this project has been mostly educational, albeit somewhat disappointing, for the author. Hopefully others may draw experience from this, and use it to improve on the results herein.

References

[1] Ahmed Bilal Ashraf, Simon Lucey, Jeffrey F. Cohn, Tsuhan Chen, Zara Ambadar, Kenneth M. Prkachin, and Patricia E. Solomon. The painful face - Pain expression recognition using active appearance models. *Image and Vision Computing*, 27(12):1788–1796, 2009. ISSN 02628856. doi: 10.1016/j.imavis.2009.05.007. URL http://dx.doi.org/10.1016/j.imavis.2009.05.007.

[2] Akshay Asthana, Michael J Jones, and Tim K Marks. Pose normalization via learned 2d warping for fully automatic face recognition. In *In BMVC*, 2011.

[3] Chih-Chung Chang and Chih-Jen Lin. {LIBSVM}: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1—-27:27, 2011.

[4] NVIDIA Corporation. CUDA Toolkit, 2015. URL https://developer.nvidia.com/cuda-toolkit.

[5] Jia Deng, Wei Dong, Richard Socher, Li-jia Li, Kai Li, and Li Fei-fei. Imagenet: A large-scale hierarchical image database. In *In CVPR*, 2009.

[6] Paul Ekman and Wallance V. Friesen. *The Facial Action Coding System.* 1978. ISBN 0931835011.

[7] Corneliu Florea, Laura Florea, and Constantin Vertan. Learning Pain from Emotion: Transferred HoT Data Representation for Pain Intensity Estimation.

[8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[9] Sebastian Kaltwang, Ognjen Rudovic, and Maja Pantic. Continuous pain intensity estimation from facial expressions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7432 LNCS:368–377, 2012. ISSN 03029743. doi: 10.1007/978-3-642-33191-6_36.

[10] T Kanade and J.F. Cohn. Comprehensive database for facial expression analysis. In *Proceedings of the 4th IEEE International Conference on Automatic Face and Gesture Recognition*, pages 46–53, 2000. ISBN 0-7695-0580-5. doi: 10.1109/AFGR.2000.840611. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=840611.

[11] Sergey Karayev, Aaron Hertzmann, Holger Winnemoeller, Aseem Agarwala, and Trevor Darrell. Recognizing Image Style. *CoRR*, abs/1311.3715, 2013. URL http://arxiv.org/abs/1311.3715.

[12] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[13] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits. URL http://yann.lecun.com/exdb/mnist/.

[14] A. Vedaldi Lenc and K. MatConvNet - Convolutional Neural Networks for MATLAB. *CoRR*, 1412.4564, 2014.

[15] University of Montreal LISA lab. *Deep Learning Tutorial, Release 0.1.* 2015.

[16] Patrick Lucey, Jeffrey F. Cohn, Kenneth M. Prkachin, Patricia E. Solomon, and Iain Matthews. Painful data: The UNBC-McMaster shoulder pain expression archive database. In *2011 IEEE International Conference on Automatic Face and Gesture Recognition and Workshops, FG 2011*, pages 57–64, 2011. ISBN 9781424491407. doi: 10.1109/FG.2011.5771462.

[17] Patrick Lucey, Jeffrey F. Cohn, Kenneth M. Prkachin, Patricia E. Solomon, Sien Chew, and Iain Matthews. Painful monitoring: Automatic pain monitoring using the UNBC-McMaster shoulder pain expression archive database. In *Image and Vision Computing*, volume 30, pages 197–205, 2012. ISBN 0262-8856. doi: 10.1016/j.imavis.2011.12.003.

[18] Andrew Ng, Jiquan Ngiam, Chuan Y Foo, Yifan Mai, and Caroline Suen. UFLDL Tutorial. *http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial*, 2010.

[19] R B Palm. Prediction as a candidate for learning deep hierarchical models of data. Master's thesis, 2012.

[20] K. M. Prkachin. The consistency of facial expressions of pain: A comparison across modalities. *Pain*, 51:297–306, 1992. ISSN 03043959. doi: 10.1016/0304-3959(92)90213-U.

[21] Ognjen Rudovic and Maja Pantic. Shape-constrained Gaussian process regression for facial-point-based head-pose normalization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1495–1502, 2011. ISBN 9781457711015. doi: 10.1109/ICCV.2011.6126407.

[22] Chengyao Shen, Mingli Song, and Zhao Qi. Learning High-Level Concepts by Training A Deep Network on Eye Fixations. In *NIPS 2012 Neural Information Processing Systems*, number 65, 2012.

[23] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, 1, 2001. ISSN 1063-6919. doi: 10.1109/CVPR.2001.990517.

A Implementation Code

The implemented code for this project and some data will be available in digital form upon request to the author.

B GPU Acceleration of CNN training

The CNN toolbox MatConvNet from `VLfeat.org` is MATLAB toolbox that is implemented efficiently in C++ and in CUDA for GPU acceleration. In order to use it, it is necessary to compile the code locally. Fortunately the toolbox contains MATLAB scripts that handle all compiling and linking of files for you.

B.1 Compiling for CPU

There are some prerequisites to be able to compile the toolbox:

- MATLAB version 2013b or higher.
- C++ compiler
  - Microsoft Visual Studio 2010 or newer
  - Alternatively one can use the Microsoft Visual C++ Redistributable. Both x86 and x64 (32-bit and 64-bit respectively) versions is needed.

This will allow you to compile the toolbox for CPU operation. Simple navigate to the directory where your toolbox is unpacked, add the *matlab* folder to the MATLAB path and run the script `vl_compilenn` with no arguments

```
> cd <MatConvNet>
> addpath matlab
> vl_compilenn
```

If compilation completes with no errors, call `vl_test_nnlayers` with no arguments to test the compiled code.

```
> vl_test_nnlayers
```

B.2 Compiling for GPU

In order to compile for GPU acceleration there are some further prerequisites.

- An NVIDIA GPU with *Compute Capability* of 2.0 or higher. The 600 series and higher should do.
- A version of the NVIDIA CUDA Toolkit (Available at [4]). The authors of MatConvNet have some recommendation for which version depending on MATLAB version at [14].

To compile for GPU, `vl_compilenn` is again called but this time with some arguments.

```
> vl_compilenn('enableGpu', true, 'cudaRoot', '/Developer/NVIDIA/CUDA-6.0')
```

This uses MATLAB build-in `mex` compiler to compile the code. The `cudaRoot` argument is optional, and tells the script where to find your CUDA toolkit. You may experience problems with this depending on your combination of MATLAB version and CUDA Toolkit. In this case you need to compile using `nvcc` (NVIDIA CUDA Complier). To do this call `vl_compilenn` like this instead:

```
> vl_compilenn('enableGpu', true, ...
               'cudaRoot', '/Developer/NVIDIA/CUDA-6.5', ...
               'cudaMethod', 'nvcc')
```

Here I did find a problem with the (as of this writing) current version of the toolbox. In the script `vl_compilenn` the arguments pertaining to the GPU compute architecture is not send to the compiler when using `nvcc`. This causes the compiler to compile with full backwards compatibility in mind, causing the process to fail, as some necessary functions are available on old architectures. I solved this by adding a line to the script (See line 305 in Listing 3)

Listing 3: Exerpt from `vl_compilenn`

```
300 % For the cudaMethod='nvcc'
301 if opts.enableGpu && strcmp(opts.cudaMethod,'nvcc')
302   flags.nvcc = flags.cc ;
303   flags.nvcc{end+1} = ['-I"' fullfile( ...
                      matlabroot, ...
                      'extern', ...
                      'include') '"'] ;
304   flags.nvcc{end+1} = ['-I"' fullfile(...
                      matlabroot, ...
                      'toolbox', ...
                      'distcomp', ...
                      'gpu', ...
                      'extern', ...
                      'include') '"'] ;
305   flags.nvcc{end+1} = opts.cudaArch ; % Correction // KN 2015-03-30
306   if opts.debug
307     flags.nvcc{end+1} = '-O0' ;
308   end
309   flags.nvcc{end+1} = '-Xcompiler' ;
310   switch arch
311     case {'maci64', 'glnxa64'}
312       flags.nvcc{end+1} = '-fPIC' ;
313     case 'win64'
314       flags.nvcc{end+1} = '/MD' ;
315       check_clpath(); % check whether cl.exe in path
316   end
317 end
```

If you have already compiled for CPU, you may experience some conflicts. If so, you need to delete the contents of the folder `/matlab/mex` in the toolbox directory before compiling for GPU.