POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Diploma research on the subject of

# Facial Emotion Recognition

POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Badania dyplomowe na temat

# Rozpoznawanie emocji twarzy

| | |
|---|---|
| Author | Roman Dubovyi |
| Study | Bachelor’s degree in Informatics in Polish-Japanese Academy of Information Technology, Warsaw, Poland |
| Specialization | Intelligent Data Processing Systems(D) |
| Student id | s13709 |
| E-mail | r_dubovoy@yahoo.com |
| First Supervisor | Sinh Hoa Nguyen, PhD.<br>Faculty of Mathematics, Informatics and Mechanics<br>University of Warsaw |
| Second Supervisor | Tuan Trung Nguyen, M. Sc.<br>Faculty of Computer Science<br>Polish Japanese Academy of Information Technology |

## Abstract

This research is addressing the problem of classifying basic human emotions (neutral, happiness, sadness, anger, disgust, surprise, fear) using facial expressions. The goal of this study is to not only have high accuracy on validation set, but to have a capability of classifying facial expressions in real-time capture.

Nowadays information becomes a very precious resource. Large amounts of data needed for those kinds of software are usually not easy to obtain. The main goal of this work is to achieve good results having small amount of data (<3000 samples).

Different neural network types with various model structures, layers, tuning parameters and input data are compared in the matter of achieving main goal. Three different architectures and approaches for this task are compared. They are: Convolutional Neural Network (CNN) with grayscale image as input, Fully-Connected Neural Network (FNN) with facial landmark as input and Fully-Connected Neural Network (FNN) with pre-processed grayscale image as input. I have to spoiler, only one of these can be accepted as successful.

This thesis contains terminology and explanations of used technologies and my assumptions regarding successfulness or failure of previously mentioned approaches.

## Abstract (polish)

W tym badaniu podjęto problem klasyfikacji podstawowych ludzkich emocji (neutralnych, szczęścia, smutku, gniewu, wstrętu, zaskoczenia) za pomocą mimiki twarzy. Celem tego badania jest nie tylko uzyskanie wysokiej dokładności w zakresie sprawdzania poprawności, ale także umiejętność klasyfikowania wyrazów twarzy podczas przechwytywania w czasie rzeczywistym.

W dzisiejszych czasach informacja staje się bardzo cennym zasobem. Duże ilości danych potrzebnych do tego rodzaju oprogramowania zazwyczaj nie są łatwe do uzyskania. Głównym celem w tej pracy jest uzyskanie dobrych wyników przy niewielkiej ilości danych (<3000 próbek).

Różne typy sieci neuronowych o różnych strukturach modelu, warstwach, parametrach strojenia i danych wejściowych są porównywane pod względem osiągnięcia głównego celu. Trzy różne architektury modelu i podejścia do tego zadania są porównywane ze sobą. Są to: Convolutional Neural Network (CNN) z obrazem w skali szarości jako wejściowym obrazem, w pełni połączoną siecią neuronową (FNN) z punktem orientacyjnym twarzy jako wejściem i w pełni połączoną siecią neuronową (FNN) z wstępnie przetworzonym obrazem w skali szarości jako wejściem. Mam spoilera, gdy tylko jeden z nich może być trafiony za udany.

Niniejsza praca zawiera terminologię i objaśnienia wyżej wymienionych technologii oraz zawiera moje założenia dotyczące sukcesu lub porażki wcześniej wspomnianych podejść.

## Contents

## List of Graphs and Images

## Abbreviations and Symbols

NN – neural network.
CNN – convolutional neural network.
FNN – fully connected neural network.
AI – artificial intelligence.
API – application program interface.
tf – tensorflow.
RELU – Rectifier Linear Unit

# Introduction

It is fair to say that AI technology is one of the most rapidly developed in the past half-decade. Those technologies already provided many solutions to previously incomputable problems.

One of them is emotion recognition using facial expressions. Solution can be used in many business applications helping to asses client's satisfaction, it can be used in crowd control and in many other examples. Of course, my goal is not this complex and ambitious. It doesn't require any deep and complicated NN model architecture, but it requires data. Every NN requires big amounts of data to generalize better. But in the case of facial emotion classification problem, due to the personalities and difference in each person's face design, it becomes even more dependent on the sample quantity.

Currently anyone can easily obtain huge datasets of about 30000 labeled images of human facial expressions. The problem with them, however, is that they are downloaded from search engine image findings and labeled using NN. Obviously, this dataset contains many false labels and even images without any feasible content. And the quality is not high, usually resolution is about 48x48 pixels.

On the other hand, it is possible to gain access to the professional dataset with images of high quality, with dedicated actors and human-made labels. Of course, such dataset will be not as big as everyone could wish. In my case I had access to <3000 images. I leave reference to them in the "List of Bibliographic References" part.

I decided to stick with smaller but more qualitative dataset. But how possible it is to create NN capable of real-time facial expressions classification using such a small dataset? Hypothetically the problem is basic, you can just throw it in the basket of computer vision and assume that it can be solved with classic approaches. But with my further investigation, described in this thesis, I replace common solution with something more unique and not necessarily more complex to achieve satisfactory result. From CNN I move to more basic FNN, even though it may look like regression, results are getting better.

Here I would also like to say about my working technology choice. For this task TensorFlow was used. It was developed by Google AI organization and the name itself describes multidimensional vectorlike structure used for computations. My choice was made on two main reasons. First – it is possible to perform computations on GPUs (Graphics processing units) that are made for operations on matrices, that speeds up training gradually, in my case making ten times faster. Second – TensorFlow has a big low-level API (Application programming interface) that lets to construct models almost from scratch and tune them as much as you want. Talking from my own experience, I can totally recommend TensorFlow for deeper understanding of AI technologies.

## CNN Theory

In the problem of emotion classification information is stored in the images in the first place. The first thought that comes to mind is using popular *CNN (Convolutional Neural Network)*.

Until middle 90's the dominating method in machine learning were *fully connected neural networks*, those have large number of parameters, which makes them costly, and they don't scale well. Then came CNN that can be considered not fully connected, each neuron of the current layer connected only to a few neurons in the previous layer, in addition weights between these neurons can be shared. This connection of convolutional layers is now called *deep learning*.

More detailed inside on how CNN works. Classical CNN consist of three components: convolutional layers, subsampling and fully connected layers. The scheme for a classical CNN looks like this:

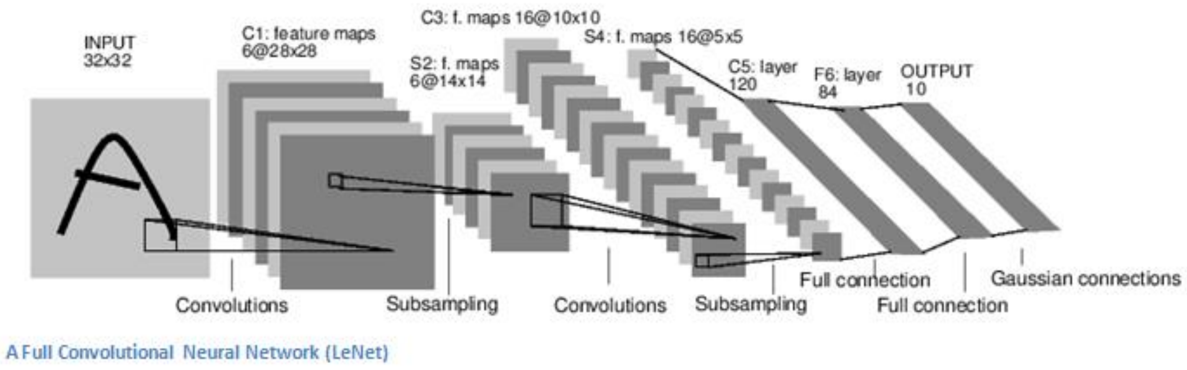
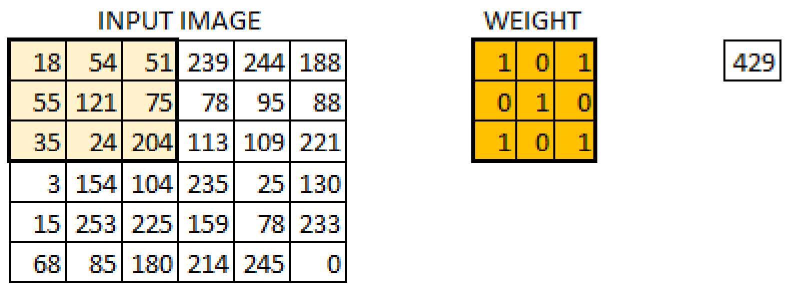


Image 1 'CNN graph example'

To understand how the convolutional layer work's, it's necessary to know components, they are: frame size, strides, number of filters, weights and activation function. You must imagine an image as the matrix with values from 0 to 255 instead of pixels. Frame of defined size (3x3 matrix on the image) traverses image matrix with the step, previously defined as stride, computing so called *convolved features.*



INPUT IMAGE

| | | | | | |
|---|---|---|---|---|---|
| 18 | 54 | 51 | 239 | 244 | 188 |
| 55 | 121 | 75 | 78 | 95 | 88 |
| 35 | 24 | 204 | 113 | 109 | 221 |
| 3 | 154 | 104 | 235 | 25 | 130 |
| 15 | 253 | 225 | 159 | 78 | 233 |
| 68 | 85 | 180 | 214 | 245 | 0 |

WEIGHT

| | | |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

| |
|---|
| 429 |

Image 2 'Convolution'

Before convolution, the filter is applied to image. Popular filters can sharpen, turn into gradients and so on. But in the CNN filters are not defined, each value is learned in the training process, highlighting different features. At the end each convolutional layer must have activation function, that tunes weights. All these components have one task - determine areas on the image responsible for classification.

After convolution frequently subsampling is applied, that again, highlights the most active zones in the input. It is done by similar operation, traversing matrix with frame (for example 2x2) that selects the biggest value. That makes matrix shrink. At the end of CNN usually common *FNN* (*Fully Connected Network*) is placed.

In theory to solve many classification problems on images, CNN would be best choice.

## CNN
## Methodology

In my research I decided to select the smaller dataset of faces. I grew confident in it by comparing the quality of contents. For comparison, I present examples:

Small dataset example (minimized x3):

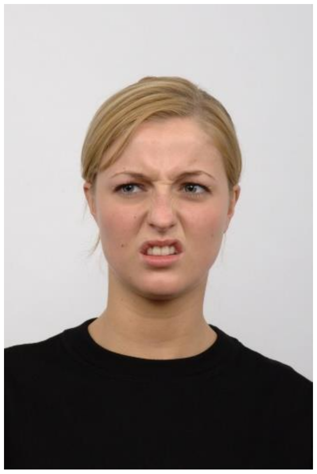


Image 3 ‘RAFD example’

Big dataset example (real size):

Image 4 ‘Bad image example’

The difference in quality is huge.

For the software to work with I chose TensorFlow. As I said, main advantages are GPU support, that speeds up process of training ten times in my case, and low-level API, that gives much more control over the model than high-level API. To be more precise high-level API are Estimators, Keras and Datasets, you can totally use only them to achieve high results. Moreover, they significantly ease the process of coding. Amongst low-level API are so called layers, which I use for my research because I wanted to understand the process deeper. In understanding the training process and debugging TensorFlow is very useful because of TensorBoard. This tool lets you visualize TensorFlow graph and training metrics fast with few lines of code.

My first steps in developing CNN were made based on LinkedIn Learning course called “Building and deploying deep learning applications with TensorFlow” (reference in the last chapter). That course makes a great introduction to NNs and TensorFlow.

The goal of NN is classification amongst exact number of classes, so for this problem supervised learning is the most suitable.

All further choices made during the implementation were based on many online papers and experiments conducted by me.

## CNN
## Technical part

At the beginning it is important to understand how TensorFlow training process works, it's all done with the so-called *Session* and *Graph*. A graph defines the computation. It doesn't compute anything, it doesn't hold any values, it just defines the operations that you specified in your code. A session allows to execute graphs or part of graphs. It allocates resources for that and holds the actual values of intermediate results and variables.

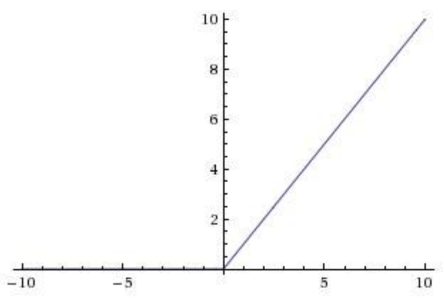
Simply speaking – every TensorFlow variables that are defined before the session will construct a Graph that will be used during Session. For easier graph construction I define layers beforehand. 'tf' stands for 'tensorflow'.

```
def convolutional_layer(input, num_input_channels, conv_filter_size, num_filters):
    weights = tf.Variable(tf.truncated_normal(shape=[conv_filter_size,
conv_filter_size, num_input_channels, num_filters], stddev=0.05))
    biases = tf.Variable(tf.constant(0.05, shape=[num_filters]))
    output = tf.nn.conv2d(input = input, filter = weights, strides = [1, 1, 1, 1],
padding = 'SAME')
    output += biases
    output = tf.nn.relu(output)
    output = tf.nn.max_pool(value=input, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='SAME')
    return output
```

Code snippet 1

Convolutional layer has 4 parameters. *Input* – variable of type tf.placeholder that contains tensors. *Num_input_channels* – variable defining number of image layers in this case, in first layer it will be 1 because input image is grayscale (in cause of RGB image number of channels would be 3), in second layer it will be the number of filters applied. *Conv_filter_size* – defines the size of a frame traversing the image. *Num_filters* – number of previously descried filters in convolutional layer, that will be changed by *Optimizator* to 'highlight' classifying features.

In this layer weights and biases are initialized. Input is passed through tf.nn.conv2d, this operation computes convolutions. It has two additional parameters. Strides – number of units as a step for moving frame, [1, 1, 1, 1] for array of images [64, 128, 128, 1] (64 grayscale 128x128 images) means frame will be moving for every image by 1 pixel in width and 1 pixel in height dimensions for 1 channel (grayscale). *Padding – 'SAME'* in TensorFlow means that padding to each image will be added in the manner that proportions will be saved, with the strides [1, 1, 1, 1] on the output we will have image of same size. After computing convolution biases are added and activation function is applied. I use the most popular activation function for deep learning called Rectifier Linear Unit, it returns 0 if receives any negative input, if input is positive it returns value back.



$f(x) = max(0, x)$

Image 5 'RELU function'

Subsampling is done using tf.nn.max_pool function, that chooses the biggest value in the frame.

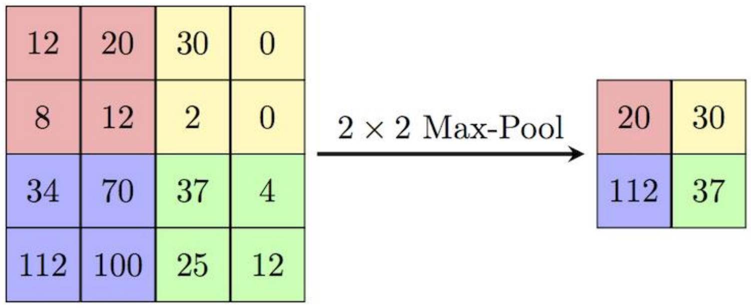


Image 6 'Max pooling'

Parameter *kzise* defines frame size, [1, 2, 2, 1] means 2x2 frame, strides work the same as in convolutional layer. With strides = [1, 2, 2, 1] frame will be moving by 2 pixels in both dimensions.

```
def flatten_layer(layer):
    return tf.layers.Flatten()(layer)
```

Code snippet 2

Flatten layer simply transforms an array of any shape into 1-D array. That let's to pass data into FNN that requires 1-D input.

```
def fully_connected_layer(input, num_inputs, num_outputs):
    weights = tf.Variable(tf.truncated_normal(shape=[num_inputs, num_outputs],
stddev=0.05))
    biases = tf.Variable(tf.constant(0.05, shape=[num_outputs]))
    output = tf.nn.relu(tf.matmul(input, weights) + biases)
    l2_loss = tf.nn.l2_loss(weights)
    return output, l2_loss
```

Code snippet 3

Fully connected layer passes input through RELU function. Additionally, I compute l2 loss for further usage.

```
def dropout_layer(input, drop_prob):
    return tf.nn.dropout(input, drop_prob)
```

Code snippet 4

Dropout layer is an old but still quite useful concept. It has only one parameter (except input), *drop_prob* – the probability with which the value will be set to 0, meaning it won't affect training process. If values sent to 0 don't affect result – those inputs will be 'turned off' in the future iterations, if they do affect result – they stay. It appears that random turns off help to increase quality of NN.

```
def batch_normalization_layer(input):
    return tf.layers.batch_normalization(input)
```

Code snippet 5

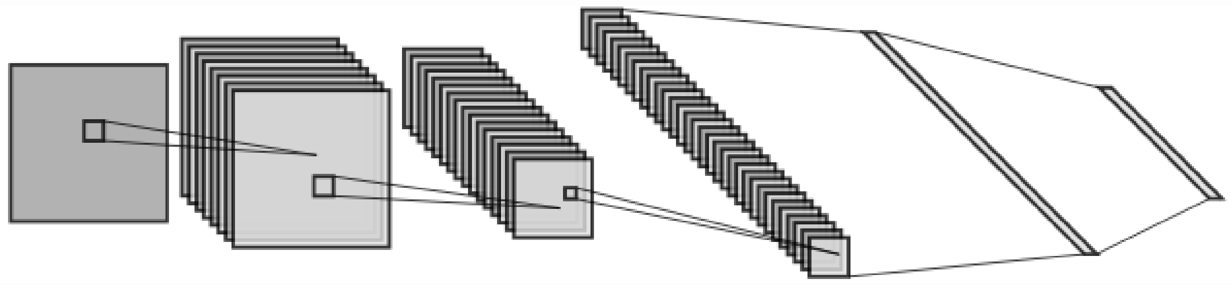
Batch normalization layer reduces high differences in values improving training process.

This is how model is defined:

```
with tf.variable_scope('convolutional_layer_1'):
    convolutional_layer_1 = convolutional_layer(input = features, num_input_channels = num_channels, conv_filter_size =
conv1_filter_size, num_filters = conv1_num_filters)

with tf.variable_scope('convolutional_layer_2'):
    convolutional_layer_2 = convolutional_layer(input = convolutional_layer_1, num_input_channels = conv1_num_filters,
conv_filter_size = conv2_filter_size, num_filters = conv2_num_filters)

with tf.variable_scope('convolutional_layer_3'):
    convolutional_layer_3 = convolutional_layer(input = convolutional_layer_2, num_input_channels = conv2_num_filters,
conv_filter_size = conv3_filter_size, num_filters = conv3_num_filters)

with tf.variable_scope('flatten_layer'):
    flatten_layer_1 = flatten_layer(convolutional_layer_3)

with tf.variable_scope('dropout_layer_1'):
    dropout_layer_1 = dropout_layer(flattern_layer_1, keep_prob)

with tf.variable_scope('fully_connected_layer_1'):
    fully_connected_layer_1, l2_1_loss = fully_connected_layer(input = dropout_layer_1, num_inputs =
flattern_layer_1.get_shape()[1].value, num_outputs = fully_connected_1_nodes)

with tf.variable_scope('batch_norm_1'):
    batch_norm_layer_1 = batch_normalization_layer(fully_connected_layer_1)

with tf.variable_scope('dropout_layer_2'):
    dropout_layer_2 = dropout_layer(batch_norm_layer_1, keep_prob)

with tf.variable_scope('fully_connected_layer_2'):
    fully_connected_layer_2, l2_2_loss = fully_connected_layer(input=dropout_layer_2,
num_inputs=fully_connected_layer_1.get_shape()[1].value, num_outputs=fully_connected_2_nodes)

with tf.variable_scope('output'):
    weights = tf.get_variable("output_weights", shape=[fully_connected_2_nodes, number_of_outputs],
                    initializer=tf.contrib.layers.xavier_initializer())
    biases = tf.get_variable(name="output_biases", shape=[number_of_outputs], initializer=tf.contrib.layers.xavier_initializer())
    prediction = tf.matmul(fully_connected_layer_2, weights) + biases
```

Code snippet 6

The NN looks like this:



Graph 1 ‘CNN model’

Input[?, 70, 70, 1]    Conv[?, 35, 35, 32]    Conv [?, 18, 18, 64]    Conv [?, 9, 9, 128]    FC[128]    FC[256]

FC – type of the layer.

[256] – shape of the output.

? – shape of dimension can be any.

Now as you can observe that in my first layer input is already defined, it doesn't need to be passed through parameters. That's because I create input pipeline using *tf.data.Dataset*. Main advantage of input pipeline is a sophisticated way of feeding data, input is divided into batches with predefined size and passed into NN gradually. That solves the problem of allocating memory for the NN. In my case, having 4gb of VRAM on video card was not enough for model initialization and input of 2600 images. That problem is present because images are put into tf.placeholder that is initialized on video card, if you choose it as the computing power for NN.

I propose to look more closely on dataset. In my case a bit more complex reinitializable dataset, it allows to change its contents so you can pass values to NN for training and validation in the same Session.

```
X, Y = tf.placeholder(tf.float32, shape=[None, number_of_inputs, number_of_inputs,
num_channels]), tf.placeholder(tf.float32, shape=[None, number_of_outputs])
A, B = tf.placeholder(tf.float32, shape=[None, number_of_inputs, number_of_inputs,
num_channels]), tf.placeholder(tf.float32, shape=[None, number_of_outputs])
```

Code snippet 7

Two pairs of placeholders, one for training set, another for validation set. Each pair has tf.placeholders for input values and labels.

```
batch_size = tf.placeholder(tf.int64)
batch_size_test = tf.placeholder(tf.int64)
```

Code snippet 8

Variable defining batch size.

```
train_dataset = tf.data.Dataset.from_tensor_slices((X,Y)).shuffle(buffer_size=100,
reshuffle_each_iteration=True) .batch(batch_size).repeat()
```

Code snippet 9

tf.data.Dataset object created using *.from_tensor_slices* to fit shapes with the input and labels. Dataset is created, then shuffled and then batched. *Repeat* at the end means that all the processes before *repeat*(creation, shuffling, batching) will be repeated each epoch.

```
iterator = tf.data.Iterator.from_structure(train_dataset.output_types,
train_dataset.output_shapes)
train_init = iterator.make_initializer(train_dataset)
test_init = iterator.make_initializer(test_dataset)
features, labels = iterator.get_next()
```

Code snippet 10

Now initializers are created. Later they can be used during the session to reinitialize Dataset to switch between train data and validation data. *Features* and *labels* can now be used directly in the model as it is shown in 'Code snippet 6'

Calculation of the accuracy, cost and optimization defined as:

```
with tf.variable_scope('accuracy'):
    labels_acc=tf.argmax(labels, 1)
    predictions_acc=tf.argmax(prediction, 1)
    correct_prediction = tf.equal(predictions_acc, labels_acc)
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
with tf.variable_scope('cost'):
    cost =
tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=prediction,
labels=tf.argmax(
        tf.cast(labels, dtype=tf.int32), 1)))
    cost = tf.reduce_mean(cost +  l2_alpha*(l2_1_loss + l2_2_loss))
with tf.variable_scope('train'):
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

Code snippet 10

Cost is calculated using cross-entropy loss, or as it sometimes called, log loss. This method is frequently used in classification problems. It measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label.

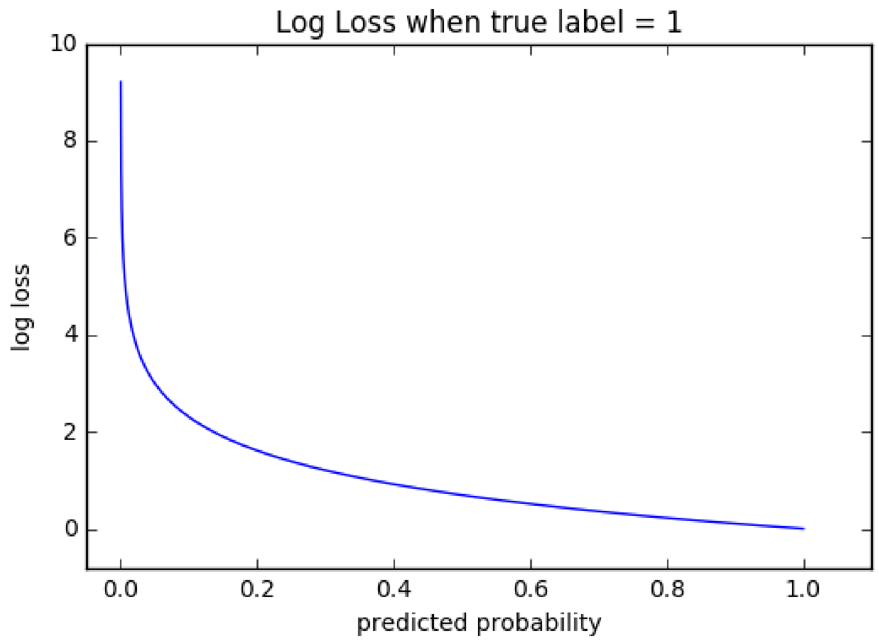


Image 7 ‘Cross-entropy function’

In case of TensorFlow it is not crucial to rescale prediction in range between (0; 1), just use tf.nn.sparse_softmax_cross_entropy_with_logits, logits are not scaled values. Later I add l2_loss from every fully connected layer to total loss to perform so called regularization. It is necessary to use it especially in cases with small dataset, regularization helps overcome overfitting and makes correction of weights more dependent on previous values. The model with regularization generalizes better.

As for optimization algorithm Adam is used. Adam is relatively new algorithm, it was presented in 2015 by the OpenAI coworker and by now Adam is the most popular algorithm in deep learning problems. The advantages are: computationally efficiency, memory requirements, suitable for problems with large number of parameters, suitable for problems with much noise. Adam currently superior to other optimization algorithms as it is shown on Image 8.

The last thing to know in technical part is how training loop is described.
The session is initialized, for every epoch optimization is done, training loss is calculated.

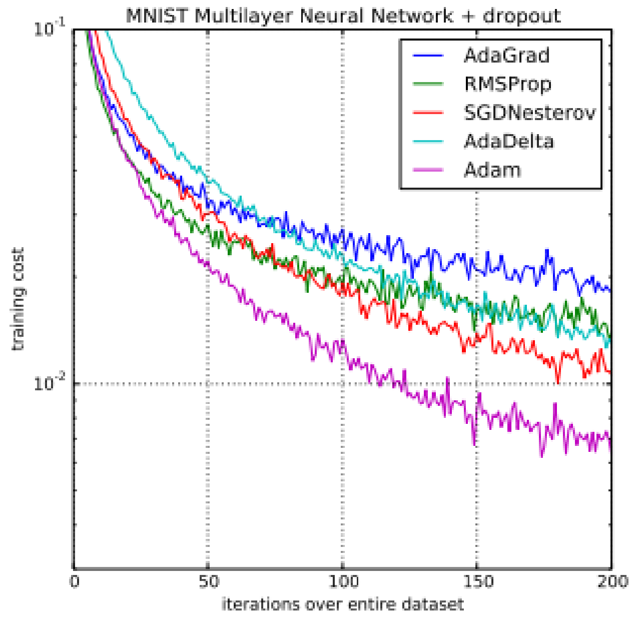


Image 8 ‘Adam optimizer comparison’

After validation accuracy and validation loss are calculated. As highlighted in code snippet 11, contents of Dataset object are switched using previously described initializers.

My stop condition is defined by the validation loss. At some point of training model starts to overfit, that makes for NN harder to generalize. When this happens, training loss continues to drop but validation loss stops in decreasing and starts increasing. “Stop when difference between present validation loss and previous validation loss is greater than some number” is what written in my code.

```
session.run([tf.global_variables_initializer()])
for epoch in range(EPOCHS):
    session.run(train_init, feed_dict={ X: train_data, Y: train_labels, batch_size:
BATCH_SIZE})
    for count in range(n_batches_train):
        _, local_loss = session.run([optimizer, cost], feed_dict={keep_prob: 1})
        total_loss_train += local_loss
    total_loss_train = total_loss_train/(count+1)
    print("EPOCH: " + str(epoch) + " cross_entropy_train: " + str(total_loss_train))
    session.run(test_init, feed_dict={A: test_data, B: test_labels, batch_size_test:
BATCH_SIZE_TEST})
    for count in range(n_batches_test):
        local_accuracy, local_loss = session.run([accuracy, cost],
feed_dict={keep_prob: 1})
        total_loss_val += local_loss
        total_accuracy += local_accuracy
    total_loss_val = total_loss_val/(count+1)
    total_accuracy = total_accuracy/(count+1)
    print("cross_entropy_val: " + str(total_loss_val))
    print("accuracy: " + str(total_accuracy))
    if epoch % 4 == 0:
        diff = total_loss_val - prev_total_loss_val
        if (diff >= 0.03):
            break
        prev_total_loss_val = total_loss_val
if (save):
    saver = tf.train.Saver()
    saver.save(session, settings['model_to_save'])
    print("training finished, model saved")
else:
    print("training finished")
```

Code snippet 11

## CNN
## Testing

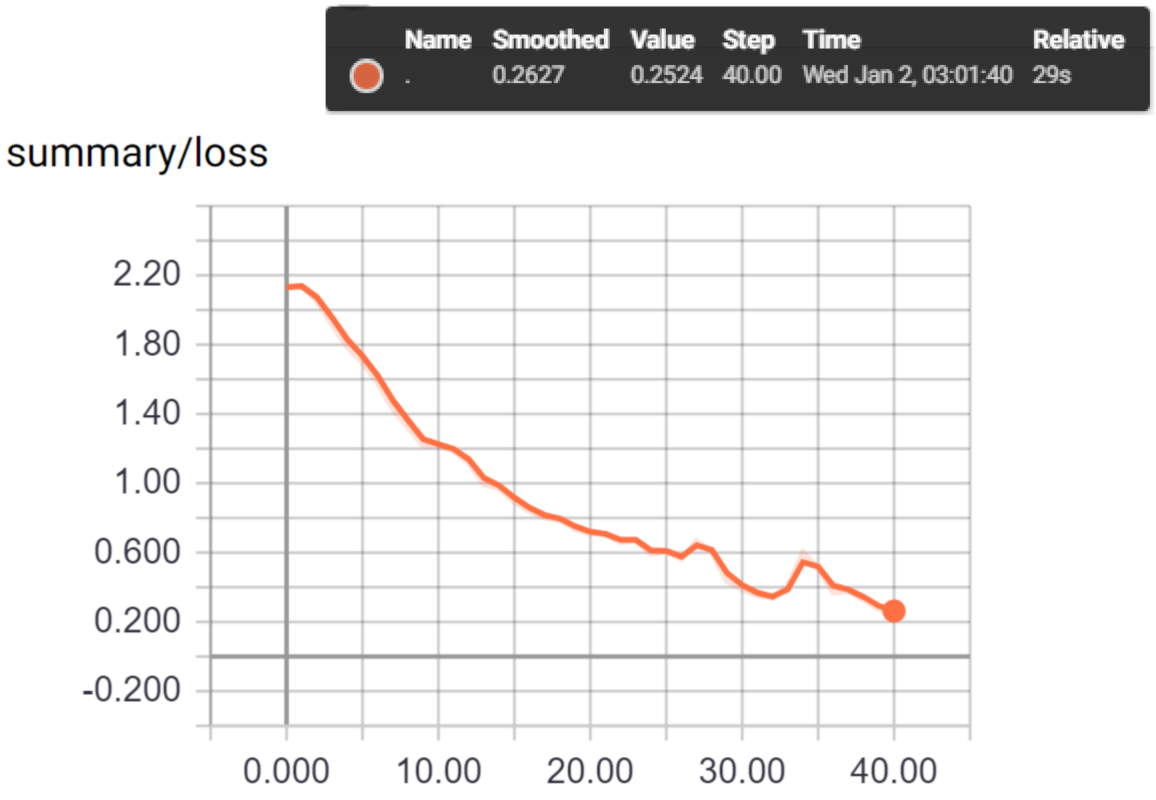
In this part performance of NN will be teste. First, I would like to have small introduction to TensorBoard. It is the suite of visualization tools that can show TensorFlow graphs, metrics diagrams and input that passes through NN. All this helps to debug and understand your NN.

All you have to do is to add values you want to visualize during each epoch. Later you can open saved with all data you have been recording.

```
with tf.variable_scope('summary'):
    accuracy_summary = tf.placeholder(tf.float32)
    loss_summary = tf.placeholder(tf.float32)
    tf_accuracy_summary = tf.summary.scalar('accuracy', accuracy_summary)
    tf_loss_summary = tf.summary.scalar('loss', loss_summary)
    merged = tf.summary.merge_all()

for epoch in range(EPOCHS):
         .
         .
         .
    summary = session.run(merged, feed_dict={loss_summary: total_loss_train,
accuracy_summary: total_accuracy})
    test_writer.add_summary(summary, epoch)
```

Code snippet 12

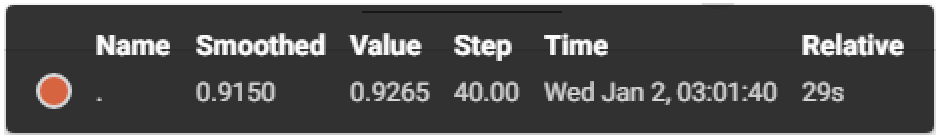
While training we can observe some good results. Only in 40 epoch model reaches the best loss value of 0.2524 before starting to overfit.



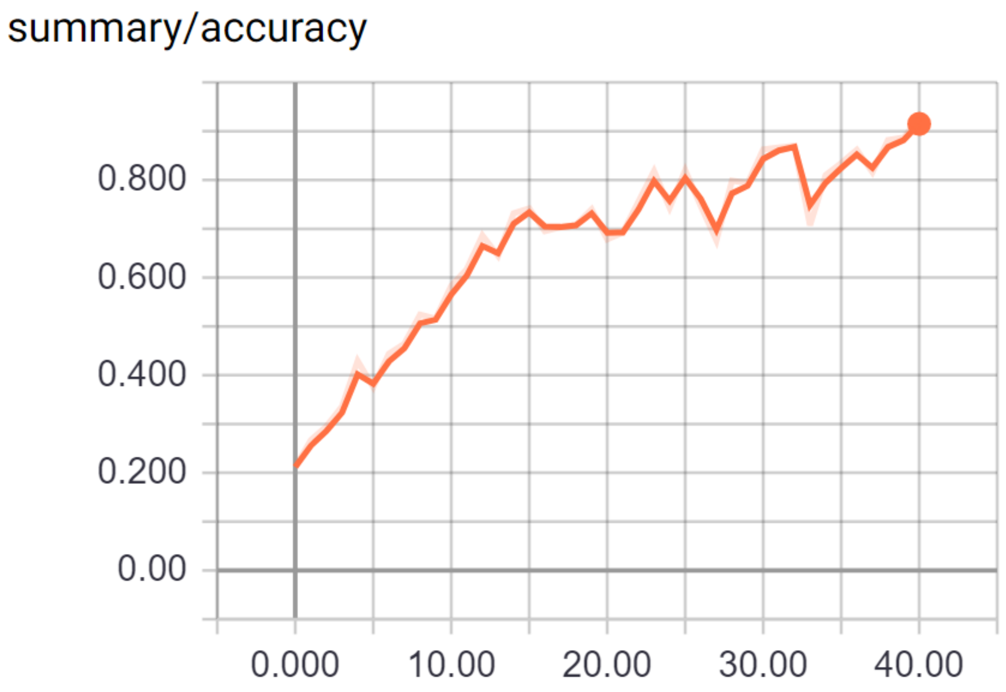
| | Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|---|
| ● | . | 0.2627 | 0.2524 | 40.00 | Wed Jan 2, 03:01:40 | 29s |

Graph 2 ‘CNN loss function’

Also this CNN has a high level of accuracy on validation set. With the best value of 92%.



| | Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|---|
| ● | . | 0.9150 | 0.9265 | 40.00 | Wed Jan 2, 03:01:40 | 29s |



Graph 3

'CNN accuracy function'

Even though accuracy on validation set was high, NN didn't pass the test in real-time emotion classification. As you can observe predictions don't match emotions.

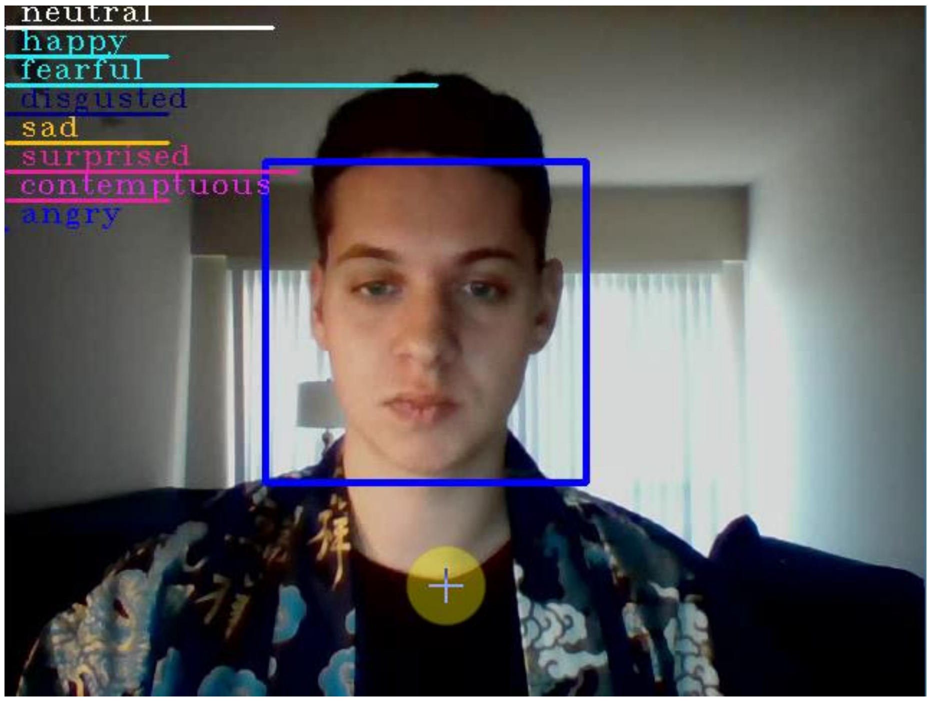


Image 9 'Neutral'

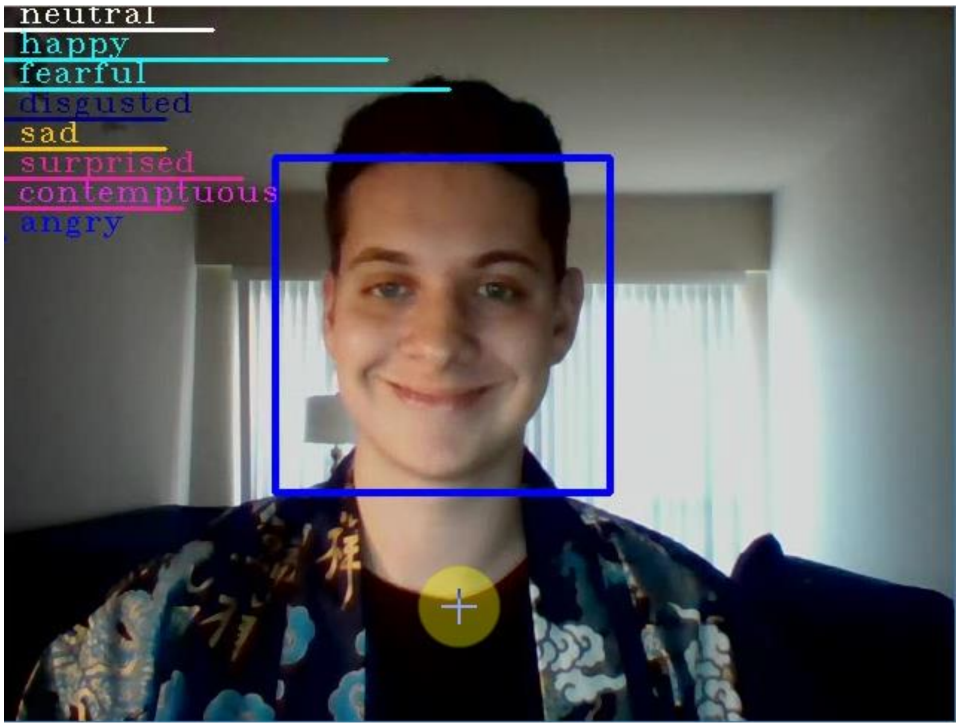


Image 10 ‘Happiness’

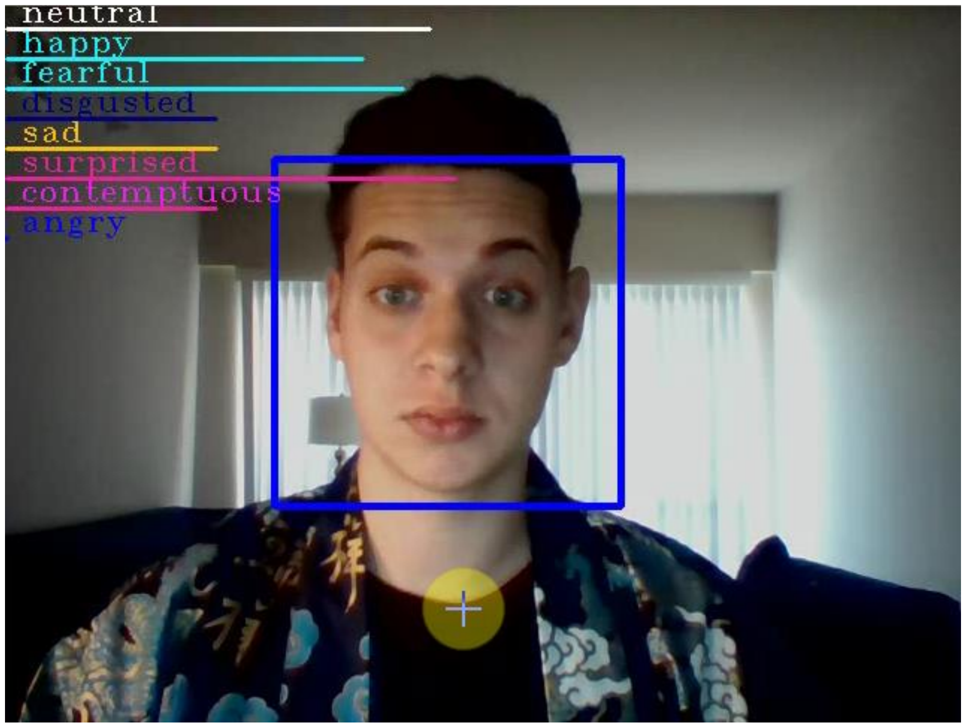


Image 11 ‘Surprise’

As it is shown on images, real-time emotion classification on my face didn’t pass the test. Even if some kind of response is visible while changing facial expressions – correct answer almost never appears. After many tries with different NN model hyperparameters and layers I failed to succeed in training reliable real-time facial expression classification CNN.

That can be of course explained by small dataset, Neural network ‘gets used’ to faces on the images. The absence of background and light conditions probably played big role too.

## FNN
## Theory

For the second attempt of creating NN on small set, one thing was clear – parameters passed into NN must be only such that have direct influence on emotion classification. What is meant is that no noisy and unwanted parameters like hair, background, shadows must not enter. Only parameters from 'emotion-determining' face parts are acceptable, amongst them: eyes, brows, mouth.

I proposed passing parameters based on *facial landmarks* (image 12). The almost absence of papers regarding emotion classification using facial landmarks made me very interested in this experiment. OpenCV provides method of extracting those landmarks. By the way, there is exactly 68 of them.

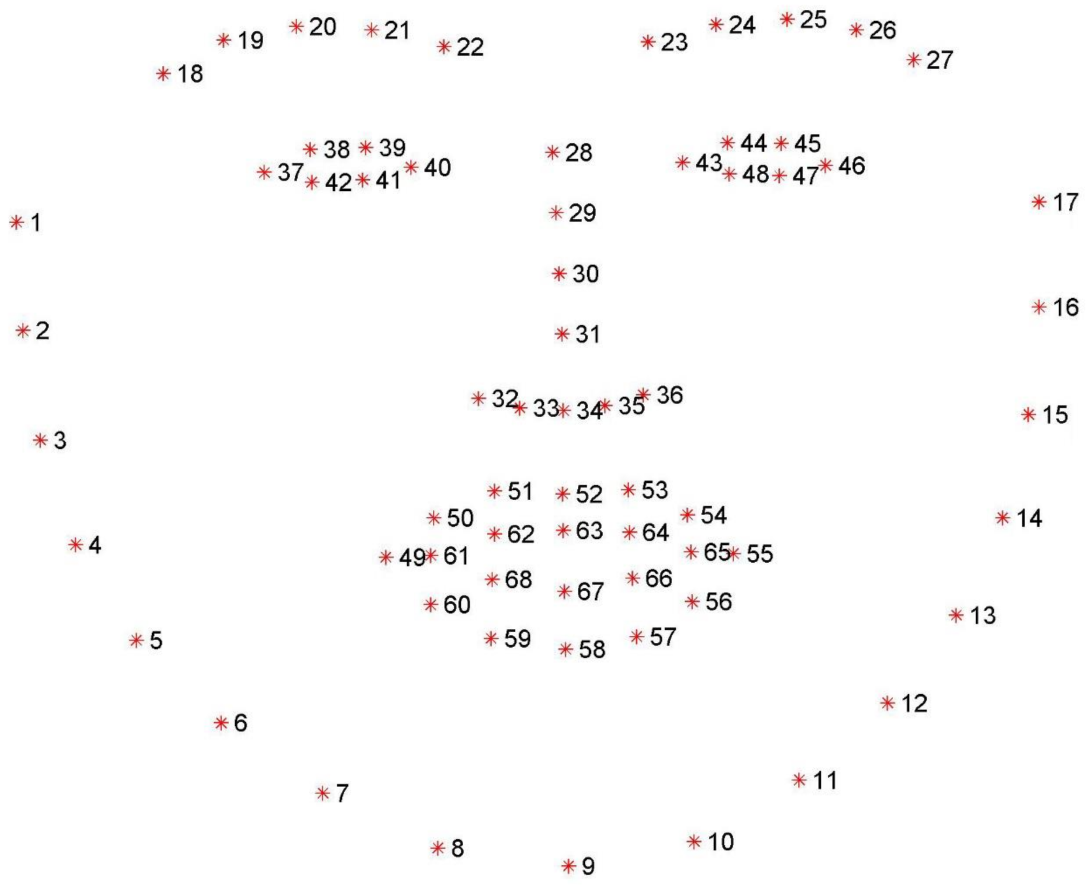


Image 12 'Landmarks'

For a problem with static number of known parameters common *FNN (Fully Connected Neural Network)* would be a good choice.

## FNN with landmarks
## Methodology

In the problem of emotion classification using facial landmarks it is necessary to understand which landmarks to pass through NN and in what form.

The first question is simple enough, by looking at the image 12 it is clear that the most ‘information bearing’ landmarks are numbers 19-27 that represent eyebrows, 47-48 that represent eyes and 49-68 that represent mouth.

But facial landmarks are just coordinate points on the image. Feeding this data would be ‘confusing’ for the NN. You can imagine that even different height of the eyebrows on a person’s face would make detection of eyebrow-defined ‘surprise’ emotion impossible. NN would never tell a difference between surprised person and person that has its eyebrows higher because of more elongated head shape. To classify emotions with landmarks it is necessary to transform them into more relative data.

Data extraction from landmarks is not complicated. The idea is to find the of a face to which all other points will be related. In the blog someone once proposed to find the center of the face by calculating the center of the gravity (or simply an average point). It is computed as easily as it sounds, average of coordinates by both X and Y axis give us the point usually located at the tip of a nose, but this approach would harm data because average point will be dependent on brows and cheeks that change while performing expressions, and a slight head rotation would change average point too. What I propose is to simply use the tip of a nose as the relative point.

The next thing to do with data is to calculate the distances between central point and other landmarks. To the lengths I propose adding angles between relative line from chin to nose tip and lines from nose tip to landmark. Here is a visual example of how mouth corner landmark is represented:

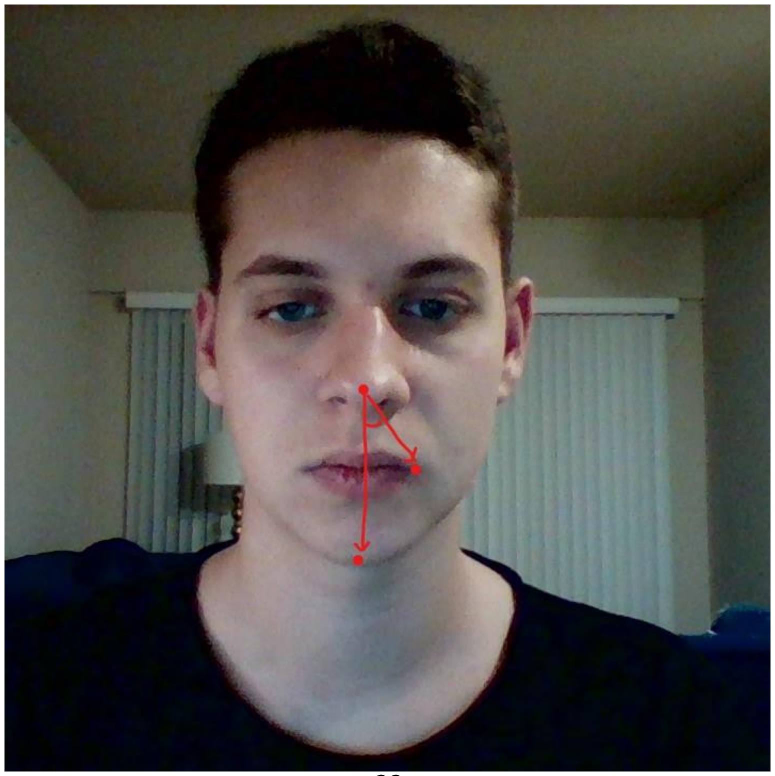


Image 13
‘Neutral’

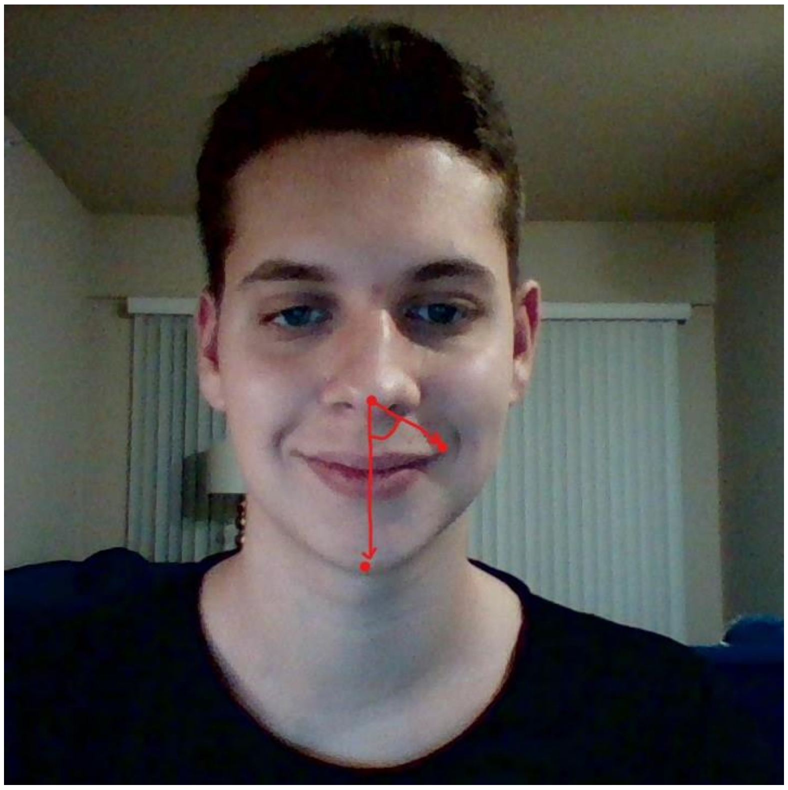


Image 14
‘Happiness’

For this exact landmark both length and angle are different between ‘Neutral’ and ‘Happy’ expressions.

What is good with this approach – it is resistant to head tilting and it doesn’t need normalization in a matter of aligning all the faces. But if coordinates are to be included to the data – it is necessary to crop and rotate all faces in dataset so that they occupy same place on image and have similar tilt angle. Example:

Image 15
‘Aligned faces’

## FNN with landmarks
## Technical part

The main part in this approach is extraction of features described previously. This useful method is contained in Dlib library for python. Landmarks detection is represented by *FaceMesh* class in my project. It has two attributes: predictor, align template, and a few functions responsible for loading images, landmark extraction and alignment of a face to the template.

```
class FaceMesh:
    def __init__(self, pred, aling_to_image):
        self.detector = dlib.get_frontal_face_detector()
         .
         .
         .
        self.ref_landmarks = self.get_landmarks_matrix(self.ref_img)
         .
         .
         .
    def get_landmarks(self, img):
        return self.predictor(img, self.ref_rect)
         .
         .
         .
    def align_single(self, img):
        landmarks = self.get_landmarks_matrix(img)
        transformation_matrix = self.transformation_from_points(self.ref_landmarks, 
landmarks)
        warped_img = self.warp_im(img, transformation_matrix, self.ref_img.shape)
        return warped_img
```

Code snippet 13

Although this is an easy and fast method of transformation. It is called FaceMesh but in the reality matrix transformations are used. Constructing real face mesh would be a costly process and would require C++ to be realized. Because of ‘primitive’ alignment image can sometimes distort. But luckily this distortion is not a problem for landmark detection.

The process of extracting angles and lengths is simple and straightforward. Add to list landmarks representing eyebrows, eyes, mouth. For each landmark calculate length to the nose tip, convert the line from nose tip to landmark into normalized vector with length equal to 1, calculate dot product between this line and normalized vertical line. After stack all the values into array.

```
for count in range(17, 27):
    x_point = landmarks.part(count).x
    y_point = landmarks.part(count).y
    x_landmarks.append(x_point)
    y_landmarks.append(y_point)
for count in range(36, 68):
    x_point = landmarks.part(count).x
    y_point = landmarks.part(count).y
    x_landmarks.append(x_point)
    y_landmarks.append(y_point)
for count in range(0, x_landmarks.__len__()):
    point = np.asarray((x_landmarks[count], y_landmarks[count]))
    length = np.linalg.norm(point - nose_point)
    lengths.append(length)
    line = [[nose_tip_x, nose_tip_y],[x_landmarks[count], y_landmarks[count]]]
    vector = get_normalized_vector(line)
    dot_product = np.dot(nose_vertical, vector)
    angles.append(dot_product)
data = np.stack((x_landmarks, y_landmarks, lengths, angles))
```

Code snippet 14

In the end each face is represented as a Numpy array with shape = [1, 42, 42, 42, 42].

For the model architecture I considered and tried many. I stopped on the model that consists of subnets for each type of information: x axis coordinates, y axis coordinates, lengths and angles. Each subnet has input, middle and output layers that are later merged into fully connected layer. Additionally, this NN has one dropout and one more hidden layer before output layer. This architecture was created with the idea that NN would first 'construct rules' for each type of parameters before they interact with each other in connecting layer. Dropout is again added to eliminate 'dead' neurons.

For implementation efficiency subnet layer is defined:

```
def subnet(input):
    weights1 = tf.get_variable("weights1", shape=[number_of_inputs, number_of_inputs],
                               initializer=tf.contrib.layers.xavier_initializer())
    biases1 = tf.get_variable(name="biases1", shape=[number_of_inputs],
initializer=tf.zeros_initializer())
    output1 = tf.nn.relu(tf.matmul(input, weights1) + biases1)
    weights2 = tf.get_variable("weights2", shape=[number_of_inputs, subnet_middle],
                                initializer=tf.contrib.layers.xavier_initializer())
    biases2 = tf.get_variable(name="biases2", shape=[subnet_middle],
initializer=tf.zeros_initializer())
    output2 = tf.nn.relu(tf.matmul(output1, weights2) + biases2)
    weights3 = tf.get_variable("weights3", shape=[subnet_middle, subnet_output],
                                initializer=tf.contrib.layers.xavier_initializer())
    biases3 = tf.get_variable(name="biases3", shape=[subnet_output],
initializer=tf.zeros_initializer())
    output = tf.matmul(output2, weights3) + biases3
    return output
```

Code snippet 15

Parameters like number of inputs, number of layer nodes, number of outputs and hyperparameters defined outside of a model:

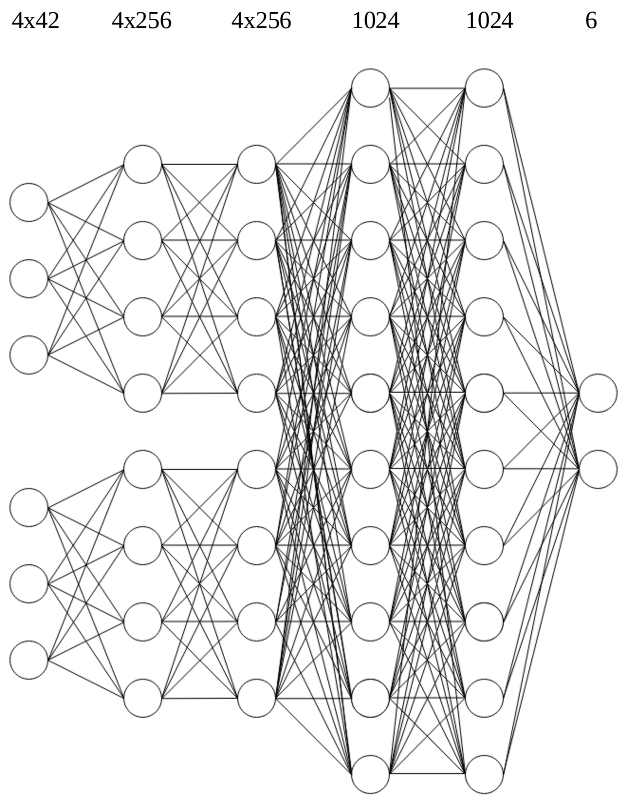
```
number_of_inputs = 42
number_of_outputs = 8
subnet_middle = 256
subnet_output = 256
layer_4_nodes = 1024
layer_5_nodes = 1024
learning_rate = 1e-4
EPOCHS = 2000
BATCH_SIZE = 32
```

The whole model definition is:

```
with tf.variable_scope('x_subnet'):
    x_subnet = subnet(features[:, 0, :])
with tf.variable_scope('y_subnet'):
    y_subnet = subnet(features[:, 1, :])
with tf.variable_scope('len_subnet'):
    len_subnet = subnet(features[:, 2, :])
with tf.variable_scope('angle_subnet'):
    angle_subnet = subnet(features[:, 3, :])
with tf.variable_scope('connected_layer_1'):
    weights = tf.get_variable("weights", shape=[subnet_output, layer_4_nodes],
                              initializer=tf.contrib.layers.xavier_initializer())
    biases = tf.get_variable(name="biases", shape=[layer_4_nodes],
initializer=tf.zeros_initializer())
    layer_4_output = tf.nn.relu(tf.matmul(x_subnet + y_subnet + len_subnet +
angle_subnet, weights) + biases)
with tf.variable_scope('dropout_5'):
    dropout_1_output = tf.nn.dropout(layer_4_output, keep_prob)
with tf.variable_scope('connected_layer_2'):
    weights = tf.get_variable("weights", shape=[layer_4_nodes, layer_5_nodes],
                              initializer=tf.contrib.layers.xavier_initializer())
    biases = tf.get_variable(name="biases", shape=[layer_5_nodes],
initializer=tf.zeros_initializer())
    layer_5_output = tf.nn.relu(tf.matmul(dropout_1_output , weights) + biases)
with tf.variable_scope('output'):
    weights = tf.get_variable("weights5", shape=[layer_5_nodes, number_of_outputs],
                              initializer=tf.contrib.layers.xavier_initializer())
    biases = tf.get_variable(name="biases5", shape=[number_of_outputs],
initializer=tf.contrib.layers.xavier_initializer())
    prediction = tf.matmul(layer_5_output, weights) + biases
```

Code snippet 16

Graphical visualization of a model would be (graph is simplified to fit in) :

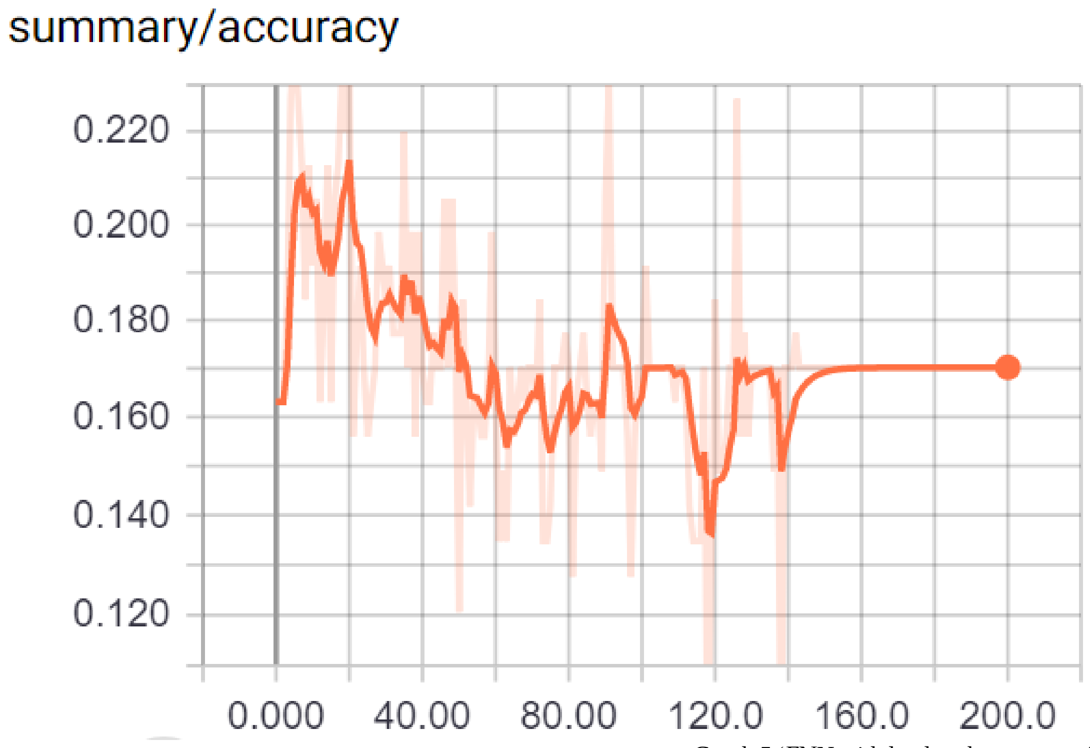


Graph 4 ‘FNN with landmarks model’

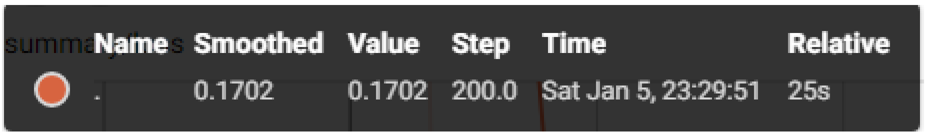
## FNN with landmarks Testing

Unfortunately, with many changes in graph, data and tuning parameters the model won't justify itself. Amongst the possible problems the most probable one is that data doesn't bear enough information. Different human faces can be too unique, and model can just not find the ratio between landmarks and emotions due to the absence of better normalization.

In this case digits on the graph are very disappointing. As you can observe, accuracy doesn't exceed 22%, I would even say it doesn't exceed 17%. Better result at the beginning of a training is due to the initialized values of weights and biases. So-called Xavier initialization is a



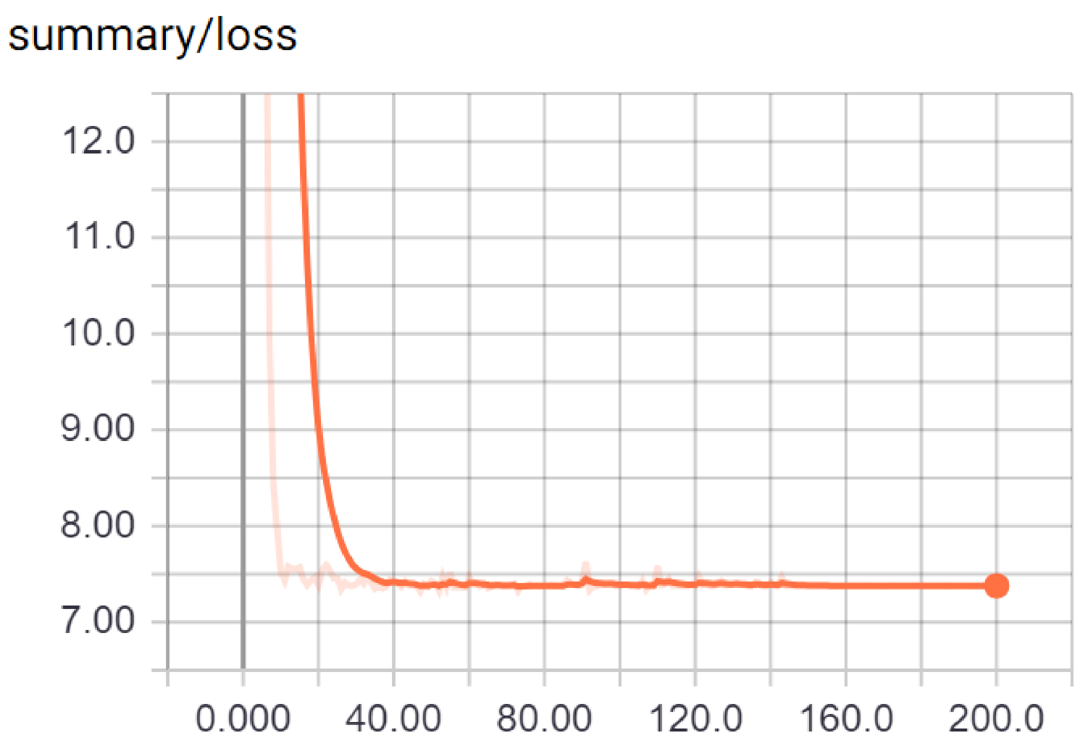
Graph 5 'FNN with landmarks accuracy function'



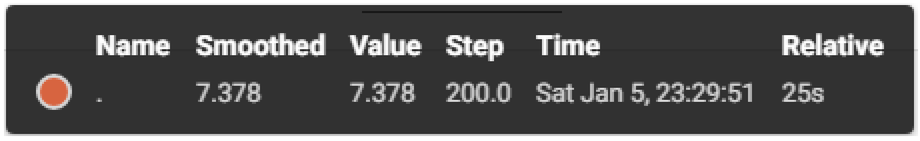
| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| ● . | 0.1702 | 0.1702 | 200.0 | Sat Jan 5, 23:29:51 | 25s |

process of setting starting values in the way that model training would be faster and more precise. So, accuracy at the beginning is just a random thing, as soon as model starts trying to fit data – it's classification capabilities weaken up to the point when accuracy is stuck at 17%.

Loss values are respectively bad. As it is shown at the 40-s epoch loss stops in decreasing and the process of training basically stops.



Graph 6 'FNN with landmarks loss function'



| | Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|---|
| ● | . | 7.378 | 7.378 | 200.0 | Sat Jan 5, 23:29:51 | 25s |

Needless to say that testing this NN in real-time has no sense.

## FNN with preprocessed images
## Theory

Let’s get back to how CNN works. I have already talked about filter parameter in CNN. Those filters are basically the equivalent for the weights in fully connected layer, but instead having only one set of weights for convolutional layer you’d better really have more of them. Each of those filters has weights that work together to ‘target’ one exact feature on the image. For example, some filter in the first convolutional layer starts recognizing top horizontal part of the image, another one can recognize light pixels. In the next convolutional layer combinations of filters from previous layer make more interesting shapes to appear. For example, those that recognize shadows, or borderlines.

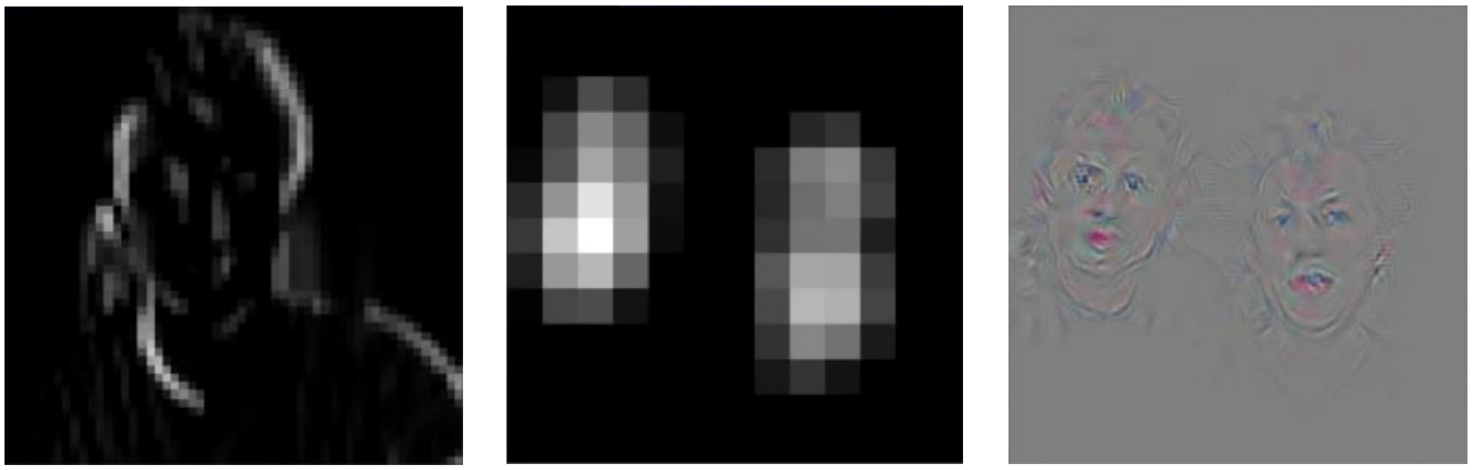

Convolution layer 1 Convolution layer 5

Image 16
‘CNN vision’

With each convolution layer filters become more specific, at the end they can recognize specific objects, like text, cats or… you’ve guessed, faces.

In the image 9 you can see an example from a video that shows what CNN ‘sees’. White pixels represent weights that are fired when image passes through. Even though one of the filters from convolutional layer 1 appears to ‘see’ a human being – that’s not true. If you look closely, it only highlights the bright edges. But in the convolutional layer 5 it looks like one filter has adapted to search for human faces. On the right-side picture you can see the image reconstructed using deconvolution.

So, in theory, why would I need to construct CNN for emotion recognition, if I can highlight all the features I need in that way myself? That would solve the problem of small dataset and will grant me even more control over situation. In theory, all I have to do is to design a filter similar to those in CNN.

## FNN with preprocessed images
## Methodology

As I already said, the principle of how CNN works inspired me to work in this direction. In this work I searched for techniques and methods of image preprocessing, that would let me get as much information about emotion on the face, as it is possible. Finally, I have found the best way of preprocessing that gave me satisfactory results. Now I would like to show the steps of preprocessing for every image that passes through my FNN.

First step is to convert image to grayscale. The thing is, that if the input of RGB image is only 3 times larger than grayscale image input, number of connections in NN should be at least in power of 3 degree bigger to work properly. In our case, however, grayscale image will let to perform edge detection.

Image 17

Second step is finding face and cropping the picture so that no redundant information is passed into NN.

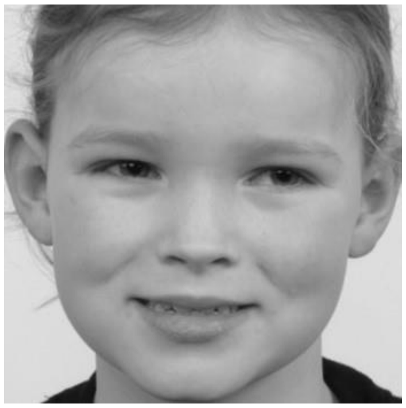


Image 18

Third step in preprocessing is to downscale and alignment with another images in the way that faces are in the same position and at the same angle. This is done in order to regularize input. It is necessary that face elements, like eyes and mouth are in the same places for FNN. Output image can sometimes be a bit distorted (not downscaled) when facial features are moved a few pixels to match the position with others. On the example you can see that the head is tilted for eyes to make almost perfect vertical line:

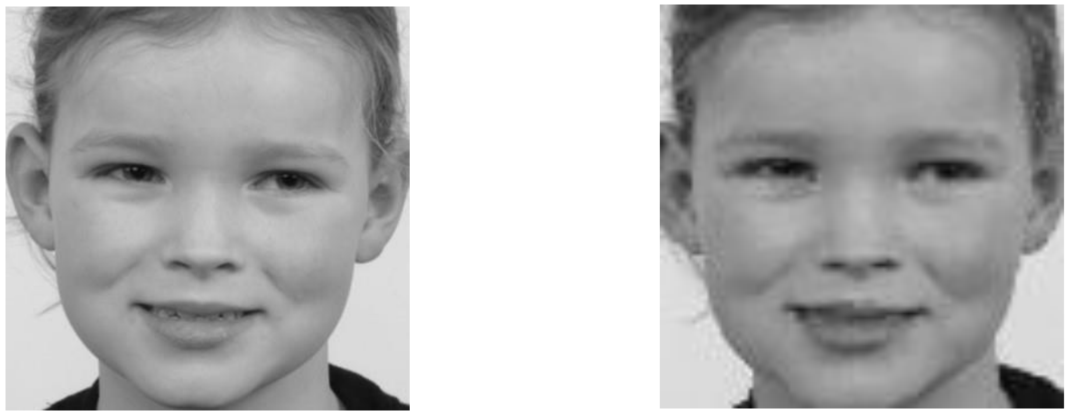


Image 19

Next step is converting into image representing gradient using Laplacian operator. Gradient represents the directional change in intensity or color in an image. This operation is supported by OpenCV library and the image will now look like this:

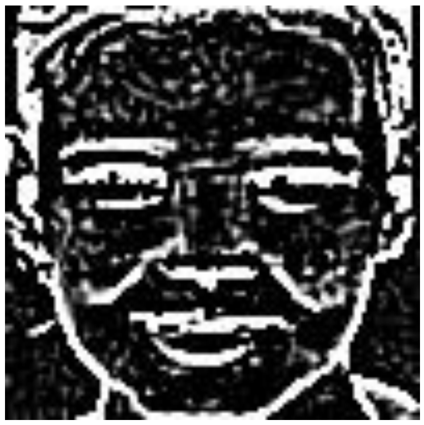


Image 20

As it is seen on the image 20 – there is plenty of redundant pixels. The last step in preprocessing is removing everything else than face

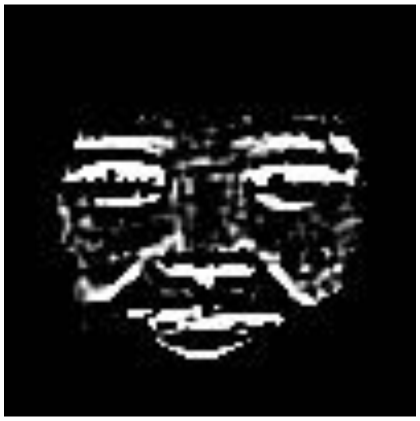


Image 21

Final image is no longer resembles it’s starting form, now it is not even possible to tell if this person is male or female, young or old, you can’t tell its skin color. But that’s awesome! NN now cannot be confused by unused features.

As you can see now, this very much resembles what convolution layer ‘sees’. But now we are sure that frame contains only those features that are responsible for emotion recognition. For this kind of input, I of course recommend skipping all convolutional layers, as we just did their job, and pass it directly into fully connected layers.

## FNN with preprocessed images
## Technical part

In this approach preprocessing of the data is the most important part. I will go through all the process of preprocessing step by step. The first step was to convert RGB image into grayscale. It is an easy but a very important part. You have two different ways to convert image into grayscale using OpenCV library.
First way is when you instantly read message as a greyscale one:

```
img = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)
```

Code snippet 17

The second one is to convert already existing image represented by *numpy.array* of a shape, for example, [240, 240, 3] into image with shape [240, 240, 1]. Each dimension represents x axis, y axis and channels respectably. In RGB you have 3 channels (red, green, blue) and in grayscale only one.

```
Img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

Code snippet 18

Next step was detecting and saving the face only. For this I have a function into which images and labels are passed.

```
Def find_faces(image_list, face_labels):
    face_crop = []
    iterator = 0
    face_cascade = cv2.CascadeClassifier(os.path.dirname(cv2.__file__) +
'\data\haarcascade_frontalface_default.xml')
```

Code snippet 19

For face detection itself OpenCV Haar Cascade classifier is responsible. It is an old but a very fast and sophisticated way of face detection. First you have to create classifier that is defined by Haar Cascade file with XML extension. This Haar Cascade defines what Classifier will be detecting. As you can imagine, Haar Cascades are also very lightweight.

```
    for x in image_list:
        image_copy = np.copy(x)
        gray_image = cv2.cvtColor(x, cv2.COLOR_RGB2GRAY)
        faces = face_cascade.detectMultiScale(gray_image, 1.15, 5, minSize=(150, 150))
        print('Number of faces detected:', len(faces))
```

Code snippet 20

Now the actual detection function is *face_cascade.detectMultiScale(img, scaleFactor, minNeighbors, minSize)*. The function has more parameters, but I use only four. *Img* – image on which detection is performed. *ScaleFactor* – Parameter specifying how much the image size is reduced at each image scale, by rescaling the input image, you can resize a larger face to a smaller one, or otherwise, making it detectable by the algorithm. *MinNeighbors* – parameter specifying how many neighbors each candidate rectangle should have to retain it. This parameter will affect the quality of the detected faces. Higher value results in less detections but with higher quality. 5 is a good value for it. *MinSize* – Minimum possible object size. Objects smaller than that are ignored.

```
        if len(faces) != 1:
            len_prev = len(face_labels)
            face_labels = np.delete(face_labels, iterator)
            if(len_prev - len(face_labels) != 1):
                print("error at iteration: " + str(iterator))
            iterator = iterator - 1
```

Code snippet 21

This part of code is invoked when in the detection output there is more than one face, which is not right, because there must be one face per image for my Dataset. Rarely classifier can detect a face where the face is not actually present. When condition is met – the last ‘face’ is not added and the label for it is removed.

```
        else:
            x, y, w, h = [v for v in faces[0]]
            cv2.rectangle(image_copy, (x, y), (x + w, y + h), (255, 0, 0), 3)
            face_crop.append(gray_image[y:y + h, x:x + w])
        iterator = iterator + 1
    return face_crop, face_labels
```

Code snippet 22

If everything is fine – rectangle, which is the output of classifier, is cut out of the picture.

I would like to mention that all face transformation operations are put in the Class called *FaceMesh* that contains *template image* with face to which all future faces are aligned. Alignment of the face is possible just with operations on matrices. It consists of three parts: extracting landmarks, getting the transformation matrix based on landmarks and referencing landmarks, and geometrical transformation on image using transformation image.

```
def align_single(self, img):
    landmarks = self.get_landmarks_matrix(img)
    transformation_matrix = self.transformation_from_points(self.ref_landmarks,
landmarks)
    warped_img = self.warp_im(img, transformation_matrix, self.ref_img.shape)
    return warped_img
```

Code snippet 23

Extraction of landmarks is supported by OpenCV. Function returns *landmarks* object that contains coordinates of each landmark on image space.

```
        Self.detector = dlib.get_frontal_face_detector()
        self.predictor = dlib.shape_predictor(pred)
        return self.predictor(img, self.ref_rect)
```

Code snippet 24

Transformation matrix is calculated based on landmarks from given image and reference image. Reconstruction of transformation matrix involves calculation of mean, standard deviation and singular value decomposition.

```
def transformation_from_points(self, points1, points2):
    points1 = points1.astype(np.float64)
    points2 = points2.astype(np.float64)
    c1 = np.mean(points1, axis=0)
    c2 = np.mean(points2, axis=0)
    points1 -= c1
    points2 -= c2
    s1 = np.std(points1)
    s2 = np.std(points2)
    points1 /= s1
    points2 /= s2
    U, S, Vt = np.linalg.svd(points1.T * points2)
    R = (U * Vt).T
    return np.vstack([np.hstack(((s2 / s1) * R, c2.T - (s2 / s1) * R * c1.T)),
                         np.matrix([0., 0., 1.])])
```
Code snippet 25

Transformation of the given image is implemented using OpenCV library's function *Cv2.warpAffine(src, M, dsize, dst, borderMode)*. *Src* – input image. *M* – 2x3 transformation matrix. *Dst* – output image with the same size as src. *Dsize* – shape of the image. *BorderMode* - pixel extrapolation method, cv2.BORDER_REPLICATE will replicate last pixels to be used as border.

```
def warp_im(self, img, M, dshape):
    output_img = np.ones(dshape, dtype=img.dtype)*255
    cv2.warpAffine(img,
                   M[:2],
                   (dshape[1], dshape[0]),
                   dst=output_img,
                   borderMode=cv2.BORDER_REPLICATE,
                   flags=cv2.WARP_INVERSE_MAP)
    return output_img
```
Code snippet 26

Last image preprocessing stage is converting image to image representing gradient and getting rid of the redundant information. For this there is a function *gray_to_gradient (images_directory, images_gradient_directory)* that converts images from directory defined by first parameter and stores them in the directory defined by the second parameter.

```
def gray_to_gradient(images_directory, images_gradient_directory):
    output = pth.Path(images_gradient_directory)
    if not output.exists():
        output.mkdir(parents=True, exist_ok=True)
    faceReader = preproc.FaceMesh("shape_predictor_68_face_landmarks.dat",
"template.jpg")
    for filename in glob.glob(images_directory):
        print('image ', filename, ' converted to grad')
        img = cv2.imread(filename, 0)
        img = cv2.resize(img, (100, 100))
        landmarks = faceReader.get_landmarks(img)
        img = cv2.Laplacian(img, cv2.CV_64F, ksize=5)
        img = remove_redundancies(img, landmarks)
        basename = os.path.basename(filename)
        basename = os.path.splitext(basename)[0]
        cv2.imwrite(os.path.join(images_gradient_directory, basename + ".jpg"), img)
```
Code snippet 27

Function *cv2.Laplacian(img, cv2.CV_64F, ksize =5)* does the conversion, key parameter is *ksize* that defines aperture size used to compute the second-derivative filters, simply speaking in the result the size of pixel representing gradients in the area is tuned with it. What else is worth noting in *gray_to_gradient*() function is that landmarks are stored before image is converted to gradient and later used to remove unused zones.

Now let's take a closer look of how I remove redundant information. The method is simple. I create the mask based on the landmarks which's border based on eyebrows, cheeks and bottom lip locations. Everything exceeding the mask is just replaced with black pixels.

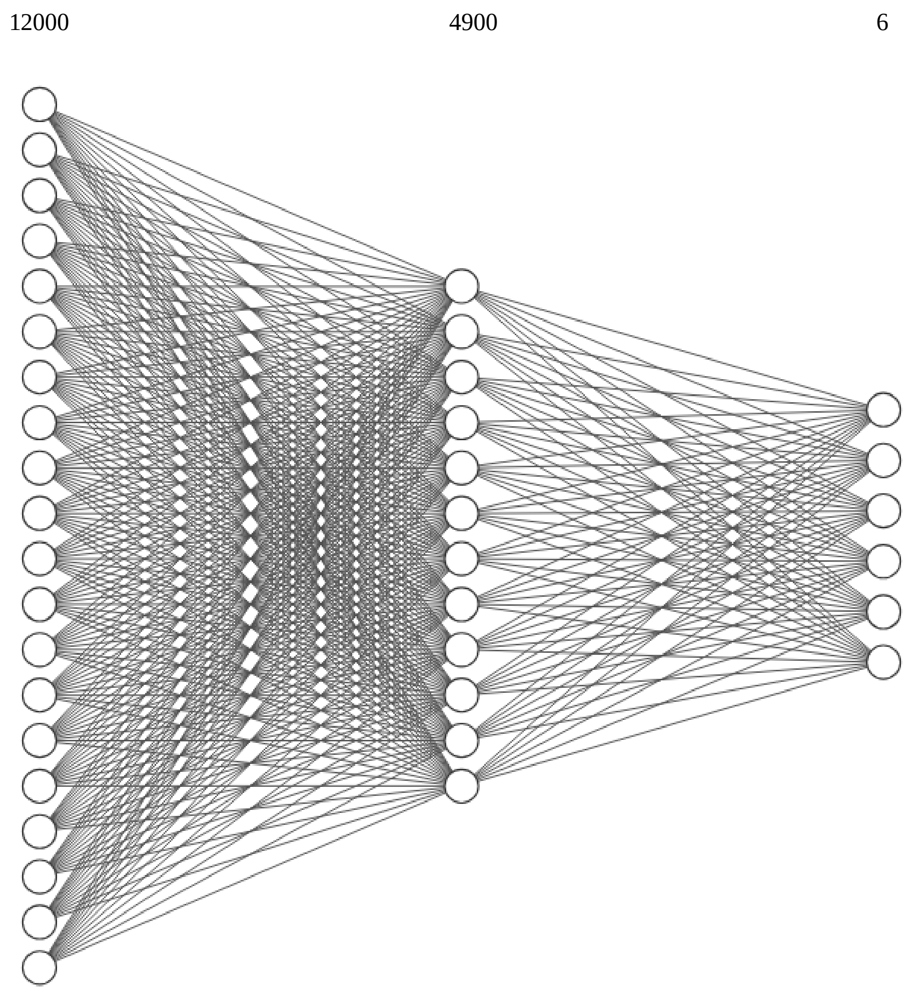
```
def remove_redundancies(img, landmarks):
    verts = [ (landmarks.part(18).x, landmarks.part(18).y), (landmarks.part(1).x + 5,
         landmarks.part(1).y), (landmarks.part(2).x + 5, landmarks.part(2).y),
         (landmarks.part(3).x + 5, landmarks.part(3).y), (landmarks.part(4).x + 5,
         landmarks.part(4).y),(landmarks.part(49).x, landmarks.part(49).y),
         (landmarks.part(60).x, landmarks.part(60).y),(landmarks.part(59).x,
         landmarks.part(59).y), (landmarks.part(58).x,
         landmarks.part(58).y), (landmarks.part(57).x, landmarks.part(57).y),
         (landmarks.part(56).x, landmarks.part(56).y),
         (landmarks.part(55).x, landmarks.part(55).y), (landmarks.part(13).x - 5,
         landmarks.part(13).y), (landmarks.part(14).x - 5, landmarks.part(14).y),
         (landmarks.part(15).x - 5, landmarks.part(15).y),
         (landmarks.part(16).x - 5, landmarks.part(16).y), (landmarks.part(26).x,
         landmarks.part(26).y), (landmarks.part(25).x, landmarks.part(25).y - 5),
         (landmarks.part(20).x, landmarks.part(20).y - 5),
         (landmarks.part(19).x, landmarks.part(19).y), (landmarks.part(18).x,
         landmarks.part(18).y) ]
    x, y = np.meshgrid(np.arange(100), np.arange(100))
    x, y = x.flatten(), y.flatten()
    points = np.vstack((x, y)).T
    black = np.zeros((100, 100))
    black.fill(1)
    polygon = Path(verts)
    grid = polygon.contains_points(points)
    mask = grid.reshape(100, 100)
    combo = black.copy()
    combo[mask] = img[mask]
    return combo
```

Code snippet 28

With these easy steps we get pretty good preprocessing that is a key to solving the problem. Last thing to discuss is the NN model architecture. As I have already said, we have just done Convolutional layers' job, so the only thing we should add into NN are fully connected layers and the output.

```
#   Strongly connected layer 1
with tf.variable_scope('layer_1'):
    weights = tf.get_variable("weights2", shape=[number_of_inputs, layer_1_nodes],
                              initializer=tf.contrib.layers.xavier_initializer())
    biases = tf.get_variable(name="biases2", shape=[layer_1_nodes],
initializer=tf.zeros_initializer())
    layer_2_output = tf.nn.relu(tf.matmul(features, weights) + biases)
#   Strongly connected layer 2
with tf.variable_scope('layer_2'):
    weights = tf.get_variable("weights4", shape=[layer_1_nodes, layer_2_nodes],
                              initializer=tf.contrib.layers.xavier_initializer())
    biases = tf.get_variable(name="biases4", shape=[layer_2_nodes],
initializer=tf.zeros_initializer())
    layer_4_output = tf.nn.relu(tf.matmul(layer_2_output, weights) + biases)
#   Output layer
with tf.variable_scope('output'):
    weights = tf.get_variable("weights5", shape=[layer_2_nodes, number_of_outputs],
                              initializer=tf.contrib.layers.xavier_initializer())
    biases = tf.get_variable(name="biases5", shape=[number_of_outputs],
initializer=tf.zeros_initializer())
    prediction = tf.matmul(layer_4_output, weights) + biases
```

Just because I have been showing graphical representations of previous NNs, I will show this one too. Although it's pretty 'standard'.



Graph 7 ‘FNN with preprocessed images model’

First layer on the graph 5 is hidden layer, where the ‘thinking’ happens. I am reminding that there is no input layer in NN because of *tf.Dataset* usage, where data is already in tensor datatype. After first hidden layer I have second hidden layer with size of 4900 units, this size is exactly the number of pixels in input images 70*70. My own experimentation shown that the layer resembling input layer connected to output has better results in solving given problem.

## FNN with preprocessed images
## Testing

You could already notice that in this FNN I had only six outputs instead of original eight. The reason behind this is that FNN gets confused by two emotions which I excluded.

The first one is ‘fear’. After some testing I made a conclusion that fear visually is very similar to emotion ‘sadness’ and almost never prediction value of ‘fear’ surpasses prediction value of ‘sadness’. To make application predict correctly in this situation, it is reasonable to add output processing, when fear will be chosen as answer with the help of additional conditions. Although for the purity of a research I removed this emotion from the set. For example, emotion on this preprocessed image is labeled as ‘fear’. I, personally, wouldn’t tell, for me it looks like ‘sadness’.

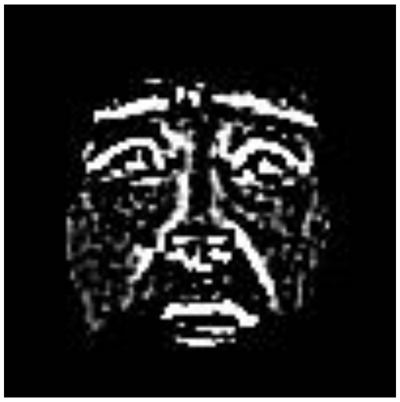
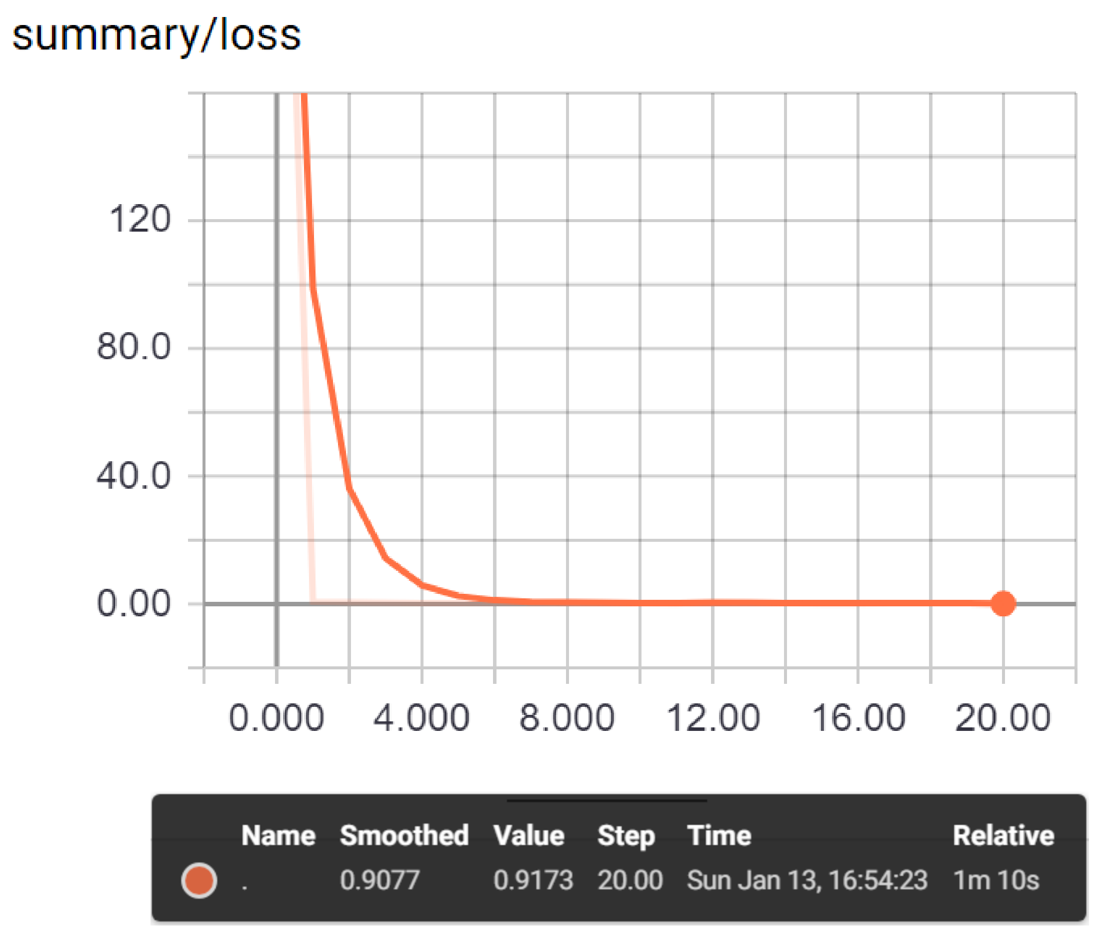


Image 22 ‘Fear looks like sadness’

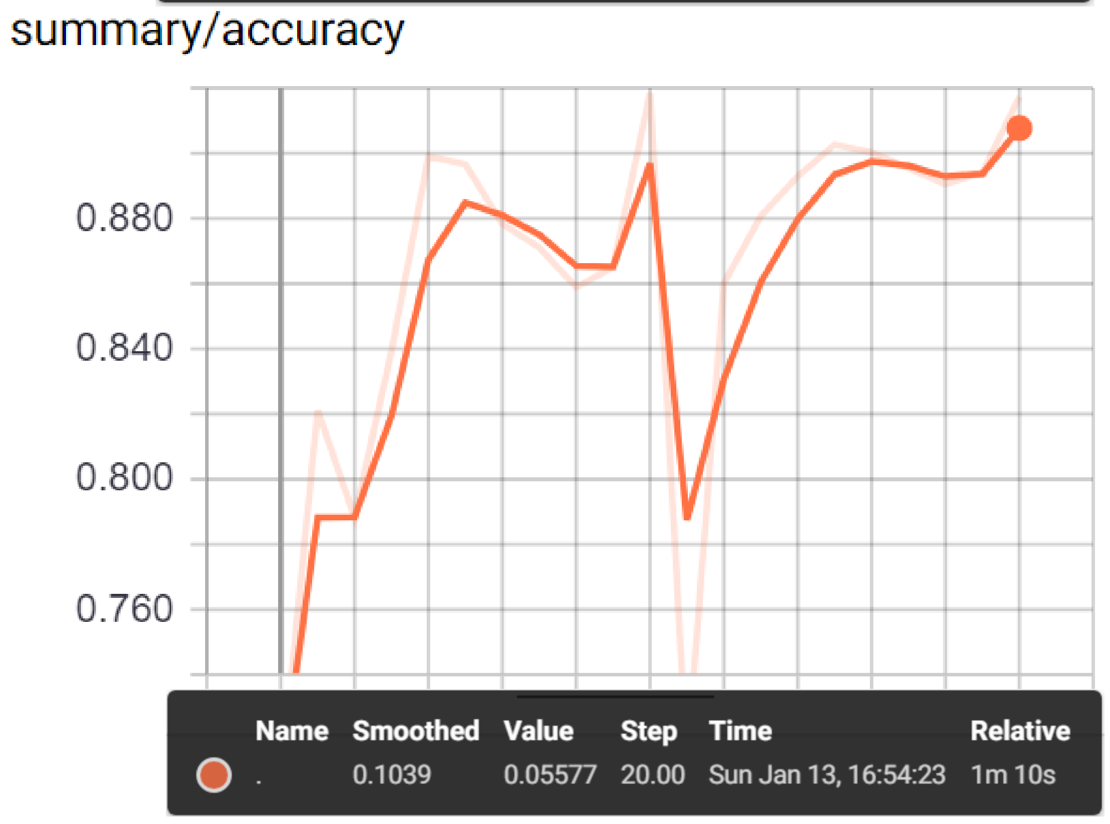
The second unfortunate emotion is ‘contempt’. I think it is obvious that contempt is a very hard and not obvious emotion. For NN, especially FNN, contempt is very hard to master because each person can express it in its own unique way. CNN with a lot of convolutions may be able to recognize different ways of expressing contempt, but FNN is more ‘general’.

Now let’s take a look at the graphs. The first one is loss. As it visible on the graph after eight iterations loss stops degreasing and the function is pretty smooth. Accuracy is high too, after the second iteration it ranges slightly with average ~80% with the top value of 91% at the end of a training.



| | Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|---|
| ● | . | 0.9077 | 0.9173 | 20.00 | Sun Jan 13, 16:54:23 | 1m 10s |

Graph 8 'FNN with preprocessed images loss function'



| | Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|---|
| ● | . | 0.1039 | 0.05577 | 20.00 | Sun Jan 13, 16:54:23 | 1m 10s |

Graph 9 'FNN with preprocessed images accuracy function'

Now the most important test – real-time capture. I test this NN, again, on myself.

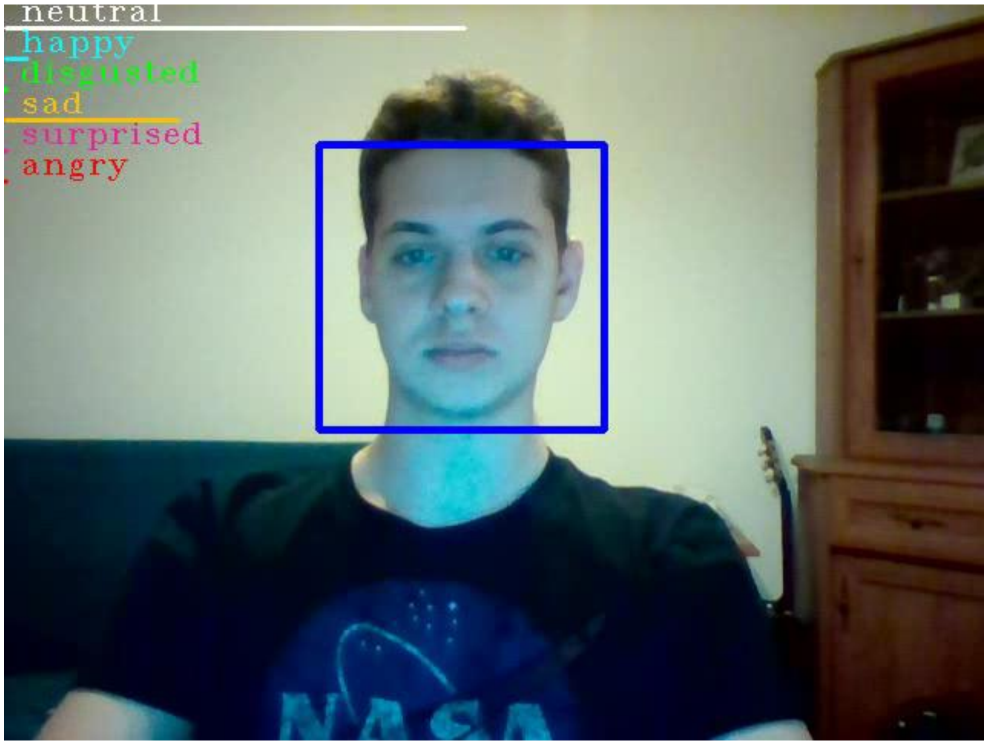


Image 23 'Neutral'

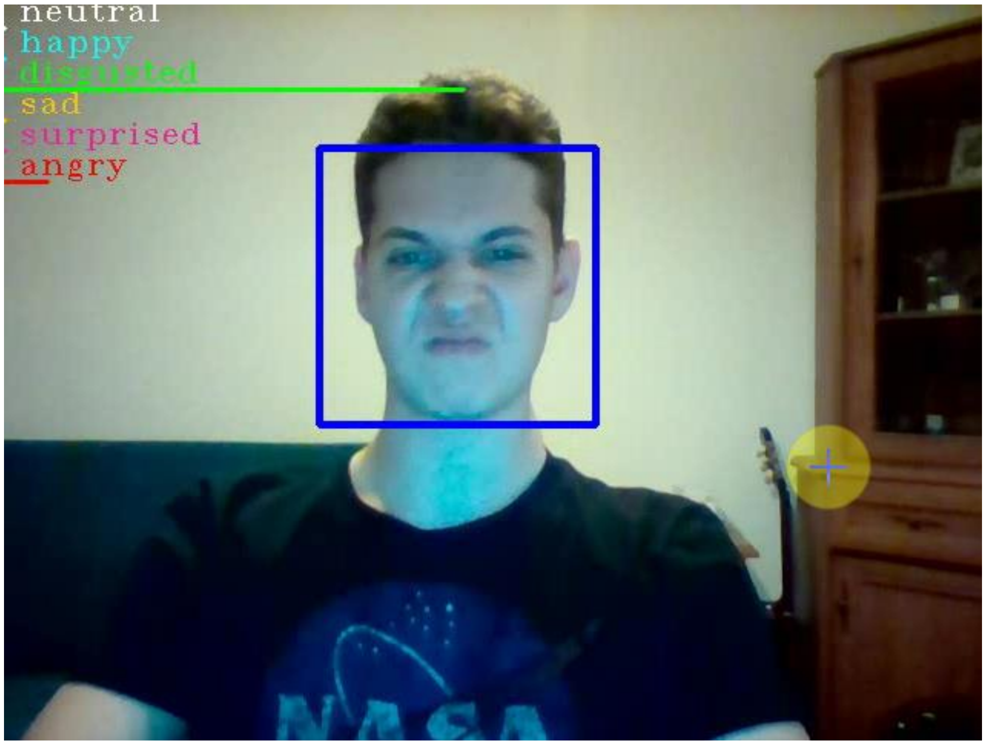


Image 24 'Disgusted'

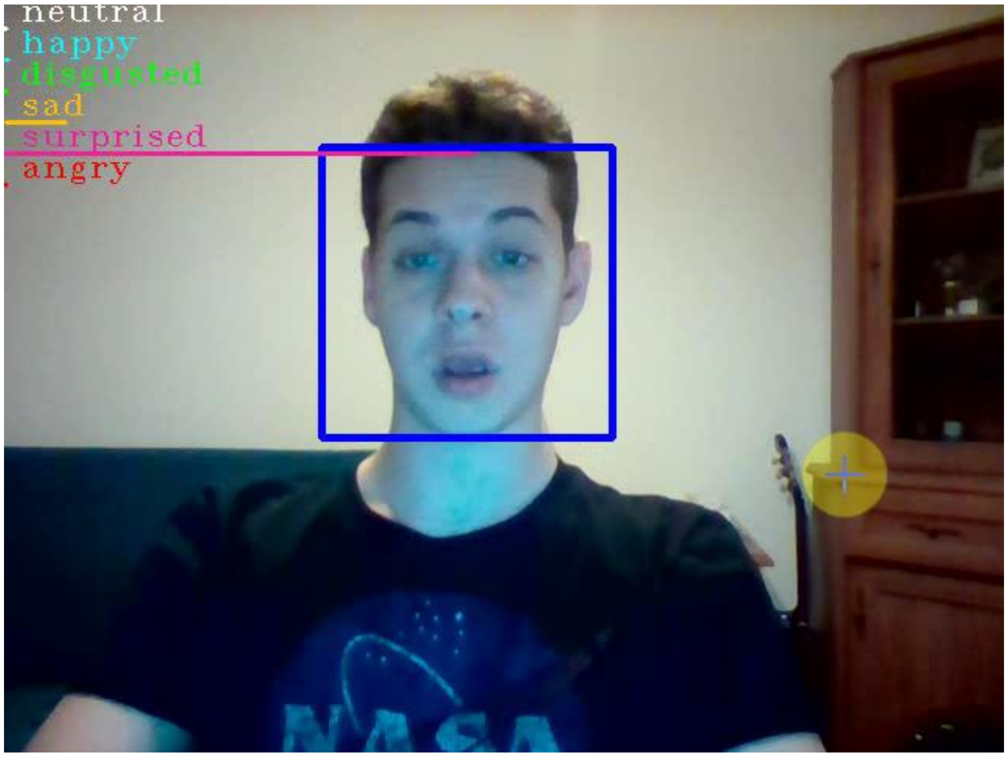


Image 25 'Surprised'

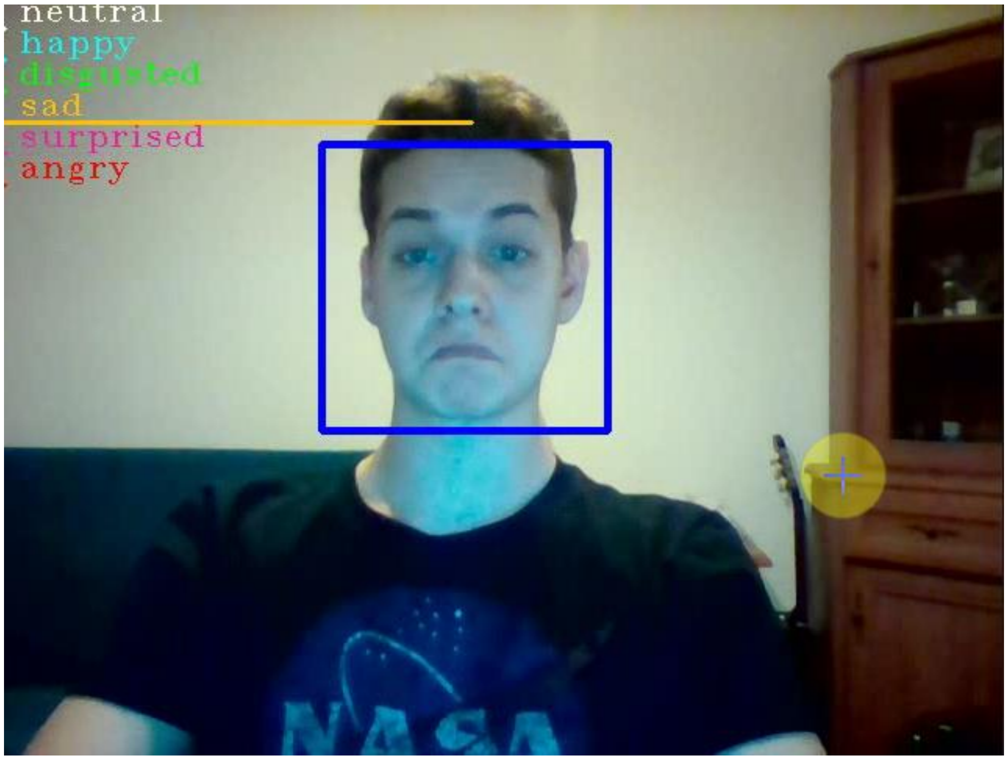


Image 26 'Sad'

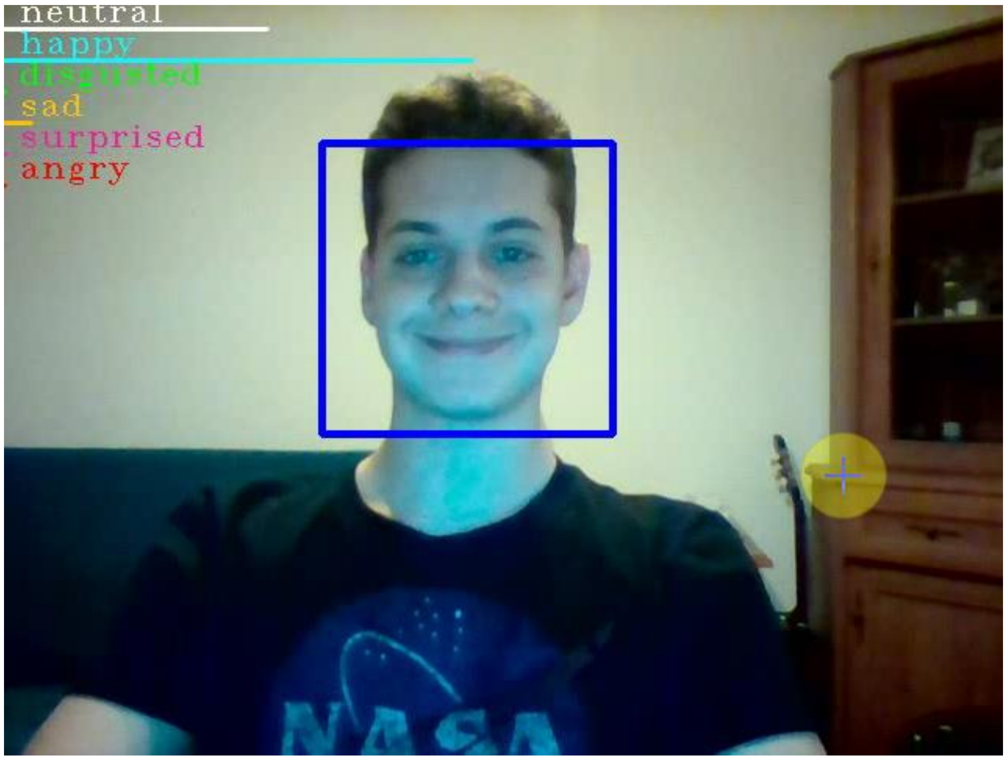


Image 27 ‘Happy’

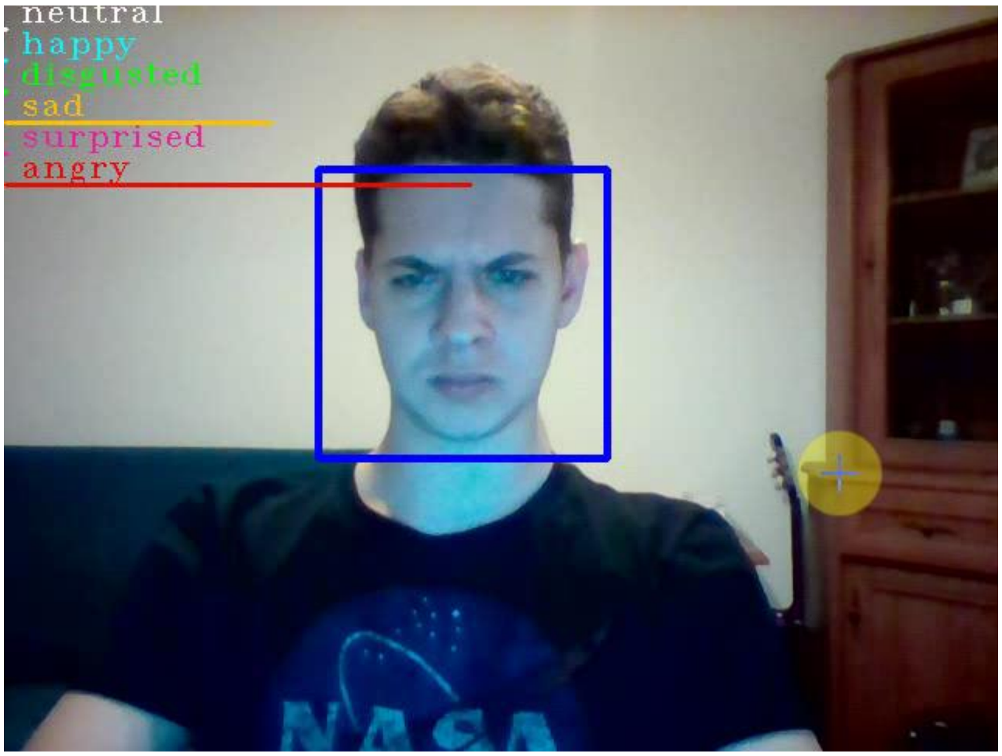


Image 28 ‘Angry’

The results are very good. Video is accessible via link.

# Conclusion

In this paper were described main approaches to my ideas and attempts of creating NNs for emotion recognition having small dataset. But except earlier described methods I had a lot of experimentation and would like to make a conclusion based on everything I went through during implementation.

First, I would like to determine the problems with CNN. In my case it has never reached desired level of sophistication when it comes to real-time testing. I concluded that my CNN isn't capable of solving this problem because it's being trained on the small amounts of images, it overfits before extracting needed features. One can say - 'why didn't you use data augmentation?'. Data augmentation is the technique of populating dataset with augmented images based on the original data. Augmentation may include resizing, rotating, adding noise, changing colors etc. I've tried that too and the only thing that comes to my mind is that problem of emotion classification is a bit harder than classifying dogs and cats. When performing data augmentation on dataset with emotions it doesn't increase training results probably because variety of those emotions doesn't change. You still have the same smiling face just with another angle, brightness or noise.

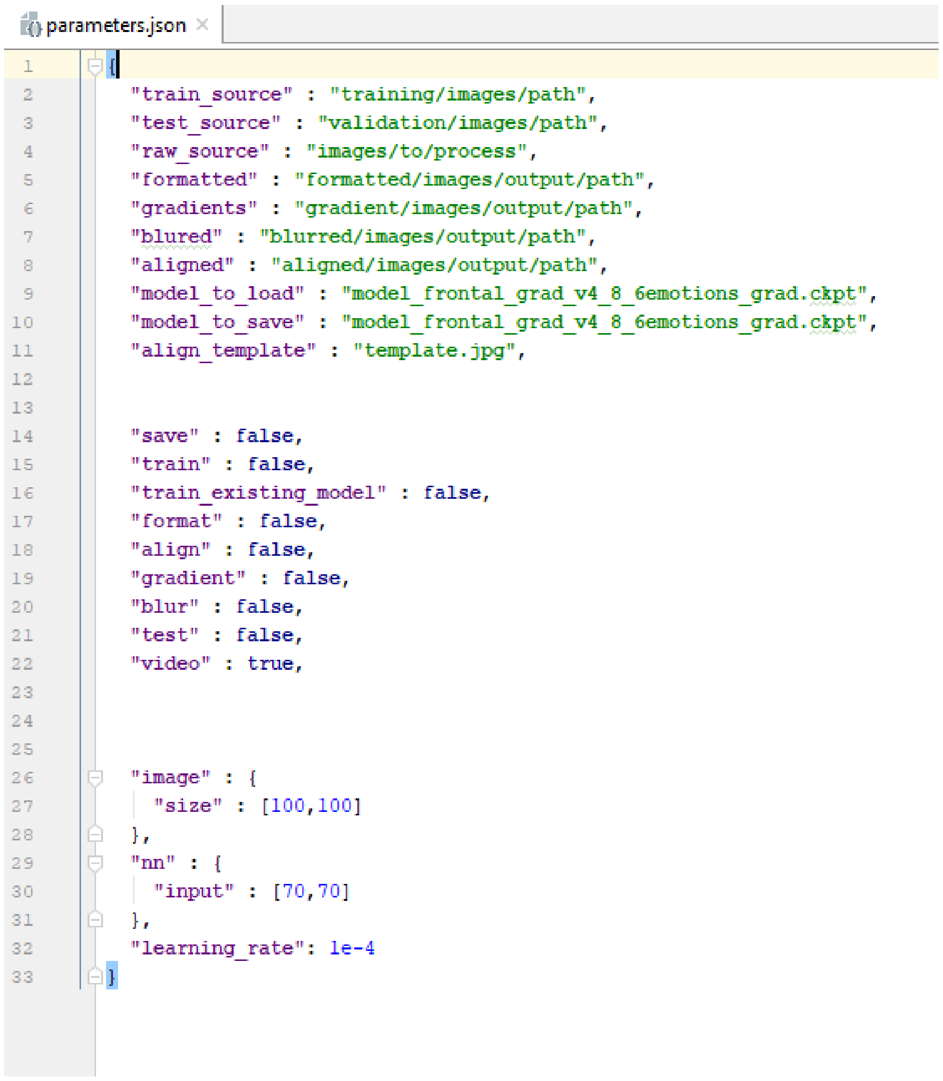
I admit I was wrong about FNN based on data from landmarks. To be fair, I had the biggest expectations for this one. With some time spent building NN and examining data the cause of failure became obvious for me. When I took the data and calculated mean and variance everything explained itself. Even if each emotion has its own mean values for angles, lengths and coordinates, the variance in all emotions was really high. What that means is that for example values of 'happiness' emotion can intersect with values for 'anger'. All of this is closely related to the fact that each person's face is very unique and measuring it in numbers probably won't make a good data for NN. What possible, though, is probably using such data for decision trees, I can imagine a good outcome for such approach.

As I already said, last FNN that is fed with preprocessed images was inspired by actual principle of convolutions in CNN. CNNs became very popular in past decade because of their speed, smaller memory usage and of course, ability to extract features responsible for determination. Of course, the last advantage is very appealing and probably that's why anyone would start solving problems with CNN. But let's take a look at Neural Networks at all. You can frequently hear people using simply phrase "Artificial Intelligence", of course it is still hard to call what we have an AI, but people still do that. Even doctors and professors do. You know why? Because we don't have a clue what exactly happens inside of NN. Some people see it as something intelligent, but of course we know that it's all defined just by using various mathematical operations, and still in some extent the part of actual 'thinking process' of NN is hidden from us. But as my example of FNN with preprocessed images shows, sometimes it's still better to perform some thinking on your behalf rather than dropping everything into NN. My final NN may lack an ability to classify fear and contempt, but still, I would like to remind that the training process took only 20 epochs. And I am proud of it.

Thank You for reading.

## How to use

In run/debug configuration input parameter ‘parameters.json’ must be defined. With this file you can control the application in some degree.



parameters.json

```
{
    "train_source" : "training/images/path",
    "test_source" : "validation/images/path",
    "raw_source" : "images/to/process",
    "formatted" : "formatted/images/output/path",
    "gradients" : "gradient/images/output/path",
    "blured" : "blurred/images/output/path",
    "aligned" : "aligned/images/output/path",
    "model_to_load" : "model_frontal_grad_v4_8_6emotions_grad.ckpt",
    "model_to_save" : "model_frontal_grad_v4_8_6emotions_grad.ckpt",
    "align_template" : "template.jpg",


    "save" : false,
    "train" : false,
    "train_existing_model" : false,
    "format" : false,
    "align" : false,
    "gradient" : false,
    "blur" : false,
    "test" : false,
    "video" : true,



    "image" : {
      "size" : [100,100]
    },
    "nn" : {
      "input" : [70,70]
    },
    "learning_rate": 1e-4
}
```

As the default trained model, I used for this paper will be predefined. You can use it to run real-time test by setting *video* parameter as *true*. You can also train models, don’t forget to change *model_to_save* parameter to prevent overriding. For image preprocessing you should use function callers: *format, align, gradient, blur* as well as input directory *raw_source* for each function and according output paths: *formatted, aligned, gradients, blurred*.

## List of References

Datasets:

Radboud Faces Database: http://www.socsci.ru.nl:8180/RaFD2/RaFD

Cohn Kanade Faces Database: http://www.consortium.ri.cmu.edu/ckagree

Literature:

Building and Deploying Deep Learning Applications: https://www.linkedin.com/learning/building-and-deploying-deep-learning-applications-with-tensorflow

Programming Computer Vision with Python 2012 – Jan Erik Solem

Computer Vision Algorithms and Applications 2011 – Richard Szeliski

Real-time test video:

https://www.dropbox.com/s/c4gwpkcl1dc1dct/fnn2_demo.mp4?dl=0

Images:

Image 1: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks

Image 2: https://www.analyticsvidhya.com/blog/2017/06/architecture-of-convolutional-neural-networks-simplified-demystified/

Image 6: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling

Image 7: https://ml-cheatsheet.readthedocs.io/

Image 8: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Image 12: https://stackoverflow.com/questions/41794191/dlib-facial-landmark-starting-index

Image 16: https://www.youtube.com/watch?v=AgkfIQ4IGaM