FACIAL EXPRESSION RECOGNITION WITH OPENCV AND DEEP LEARNING

Andrew Showers, Jingyu Su, Yanshan He

ECSE-6610 PATTERN RECOGNITION SPRINT 2017 CLASS PROJECT

Table of Contents

1. Introduction

As the complexity in computer software increases, user expectations have grown as a result. In an attempt to match these growing expectations, software will need to expand in functionality in order to interpret and react to the environment. One such way is through facial recognition as a method to estimate user emotion. These detections will have to be performed in real-time to provide an interactive experience for the user. This type of classification problem has been researched in the past, with successful examples in support vector machines (SVM) [1], neural networks (NN) [2], and distance classifiers [3].

In a study performed by Ghimire et al. [1], an attempt was made to classify facial expressions through SVM and NN using the Cohn-Kanade image set as training data. Given the original dimensions of the data set using the raw pixels of the images, the NN was able to achieve 92.29% accuracy and the SVM was able to achieve 93.80%. However, they found that by applying a local binary pattern (LBP), they were able to improve accuracy to 96.22% for NN and 95.24% for SVM. This type of feature transformation will be discussed later, but this study shows the benefits of using alternative representations of the data set. Furthermore, by implementing Gabor wavelets for dimension reduction, accuracy slightly improved for both SVM and NN.

As for neural networks, a recent study be Levi and Hassner [2] show the effectiveness of implementing a convolutional neural network (CNN) in order to classify emotions. The CASIA webface data set was used in order to train the neural network. Model accuracy was tested using both the original RGB images as well as LBP representations. The results showed that LBP performed slightly better than the raw image data, similar to the findings by Ghimire et al. [1]

Another approach to emotion detection is to use Kullback-Leibler (KL) divergence and LBP. Work performed by Vupputuri and Meher [3] have shown the high accuracy can be achieved by applying a KL divergence distance classifier to LBP facial features. Using the JAFFE database, they were able to achieve an accuracy of 95.24%. A benefit to this approach is the ability to implement this classification algorithm in high speed applications, whereas SVM would struggle to compete due to complex training. Furthermore, neural networks lack consistency and require thorough experimentation.

Taking these studies into consideration, we will attempt to achieve similar accuracy in a real-time system. We will implement a variety of classification algorithms, including SVM, CNN, and K-nearest neighbor. Furthermore, given the wide-spread use of LBP and its success in emotion recognition, we have decided to investigate whether this representation improves our classification results. Since we wish to apply this algorithm in real-time, we will need to find a classifier that is capable of processing multiple samples a second while still retaining a high level of accuracy. Furthermore, this real-time data acquisition will allow us to test the generalization of the model by providing unique images taken under different conditions compared to the training data set.

2. Data set

The data set we used for training is the Cohn-Kanade database, released on September 28th, 2010. There are 593 sequences across 123 subjects. All the pictures are classified as one of seven emotions. These seven emotions are Neutral, Anger, Contempt, Disgust, Fear, Happy, Sadness, Surprise. While the diversity in the classification labels allow us to experiment with a difficult classification problem, an issue remains that even by the human eye some of the classifications are ambiguous. An example of one of the original samples can be seen in Fig. 1.1.

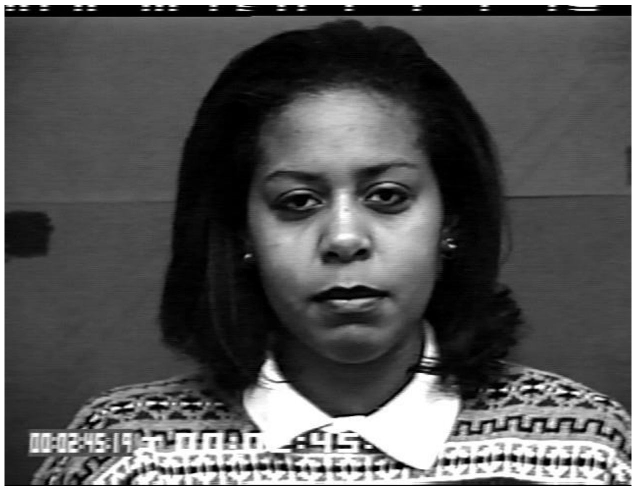


Figure 1.1: *Original picture*

All the images are captured from a video. This poses as a problem since one emotion is captured over a collection of frames, some of which are not clear in the emotion being expresses. For example, Fig. 1.2 shows 2 sets of images labeled as Disgust and Happy. It is difficult to tell what expression is in the first few pictures. Additionally, it is difficult to distinguish if the first set of images are angry, disgust or contempt.

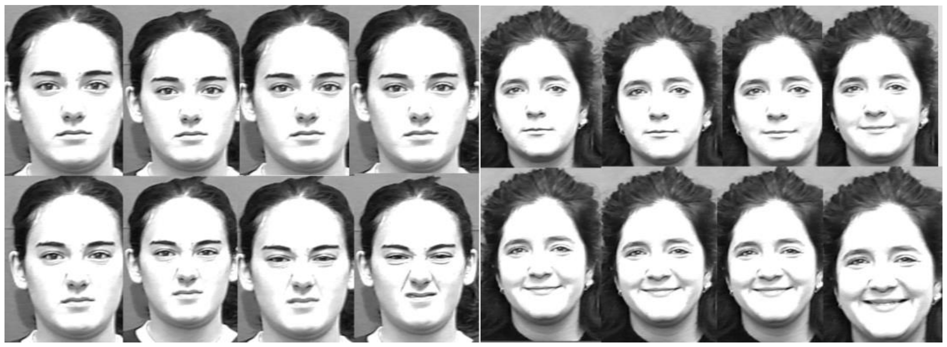


Figure 1.2: *Fixed emotion*

3. Image Preprocessing

Before implementing a classification algorithm on a dataset, it is important that the data has a level of consistency. This consistency is also important for real-time sample processing as the samples should be in a similar format as the training data. We decided to apply consistency in the following areas:

- Color-Depth
- Face Position / Cropping
- Dimensionality

In order to achieve the same color-depth, all images from the dataset were passed through a grayscale filter in OpenCV. This was crucial as the original data was gathered over many years with some in black and white and others in full color. This same grayscale filter is applied in the real-time application as well to ensure test samples resemble the training samples.

We then used a face detection algorithm, specifically a Haar-cascade classifier, to determine where a face is in an image. Upon detection, we cropped the face out to ensure that other information, such as background objects, has little impact on the overall image. This is an ideal solution since the most relevant information in the image is the features of the face. We applied this face detection algorithm to our original dataset of 5,876 images. The detection algorithm was very successful with only 1 image that failed to capture a face. This same classifier is used to detect faces in real-time using a video camera to capture samples, so by cropping out the face in this manner we have ensured positional consistency in both the training set and real-time application.

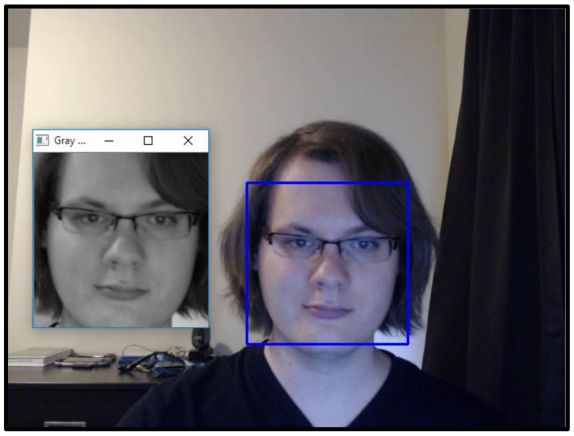


Figure 3.1: *Real-Time video capture with facial recognition and image processing.*

As for dimensionality, resizing the dataset through OpenCV was a simple task which worked for both preprocessing and real-time sample gathering. However, choosing the appropriate image size is not as simple. Larger images, such as 200x200, provide a high level of detail at the cost of requiring 40,000 dimensions, 1 dimension for each pixel. Although we implemented ways to reduce this dimensionality, the best solution ended up being a 32x32 image size fed through the neural network. This gave us a dataset with 1,024 dimensions which was large enough to contain enough information for a successful classification while still being small enough to process through the SVM and neural network classifier.

Sample Size

Give the original dataset of 5,876 images, we faced an issue where this would not be large enough to classify 7 emotions in a neural network. Since we were unable to find other datasets, we were faced with 2 options. Either expand the dataset by adding in new samples ourselves or modify the original images to create new unique samples. With a goal of acquiring 50,000+ samples, manually adding to the dataset was not practical. Instead we took the original images and applied various forms of image filters and noise in order to acquire new images that to the human eye were roughly the same but pixel by pixel values were entirely unique. A few examples can be seen below:

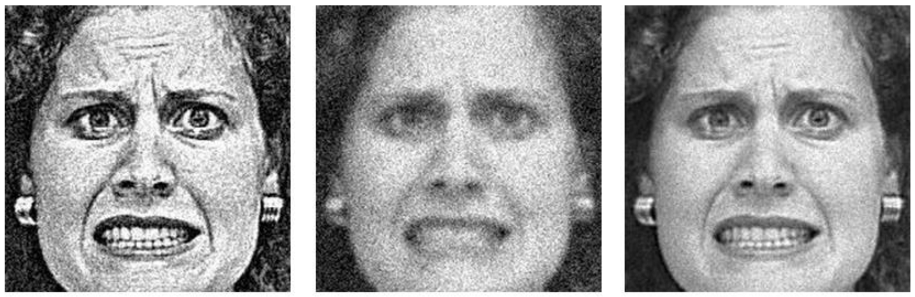


Figure 3.2: *Noise is applied to a source image to create unique samples.*

The algorithm used to generate these images used a combination of image filters from the Python Image Library (PIL) and Scikit-image. The sample size was increased from 5,876 to 94,016, well above our original goal. Algorithm 1 demonstrates the process in which noise is applied.

```
blur_radius = [0, 0.5, 1, 1.5, 2]
noise_type = ["gaussian", "poisson", "speckle"]
for b in blur_radius:
    for n in noise_type:
        Apply Gaussian Blur with radius = b                  (PIL)
        Apply Detail Filter                                  (PIL)
        Apply Sharpen Filter                                 (PIL)
        Apply Unsharp Mask with random radius: 1, 3, 5       (PIL)
        Apply random noise filter with type = n              (Scikit-image)
        Save New Image
```

Algorithm 3.1: Implementing noise as a means to increase training size and improve model robustness

While this algorithm provided us with images that were unique from each other, there is still room for future improvement. These filters adjust pixels values but positional patterns still remain due to the lack of rotational warping. Applying small rotations in combination with distortion filters would give even more unique samples while still retaining the original classification.

Local Binary Pattern

A popular approach for facial expression recognition is to transform the data via a local binary pattern algorithm.[5]This transformation helps measure the intensity changes in a pixel in comparison to its surrounding neighbors. This can provide useful information such as edges which will be less susceptible to change given slight lighting variations. By implementing this algorithm, the resulting dataset and real-time sample gathering will be more robust and provide accurate classifications under non-ideal settings.

The local binary pattern feature vector is determined by going over each pixel and taking the pixel intensity as a threshold. Once the threshold is set, each neighboring pixel can be valued at either a 0 or 1 if their own intensity exceeds that of the threshold. These values can be concatenated together in binary form to create a single representation for the center pixel. Determining how many points used as neighbors for a given pixel can be determined by two parameters, number of points to sample and the radius of circle. Fig. 3 shows a few possible configurations.

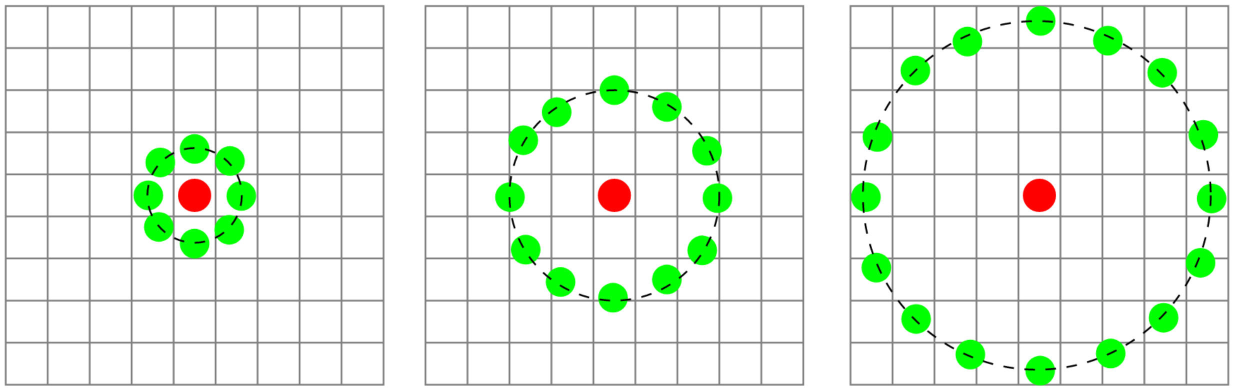


Figure 3.3: Example of various neighborhood configurations. Source image courtesy of Xiawi: https://commons.wikimedia.org/wiki/File:Lbp_neighbors.svg

In our approach, we used a radius of 3 with 24 sampling points. Originally we had written a Python function to perform this algorithm but it was too slow to be used in real-time as each image would take roughly a second to process. As an alternative, we took advantage of the Scikit-image library which is capable of processing images and returning a local binary pattern representation in under a tenth of a second. An example of the image transformation can be seen in Fig. 3.4.

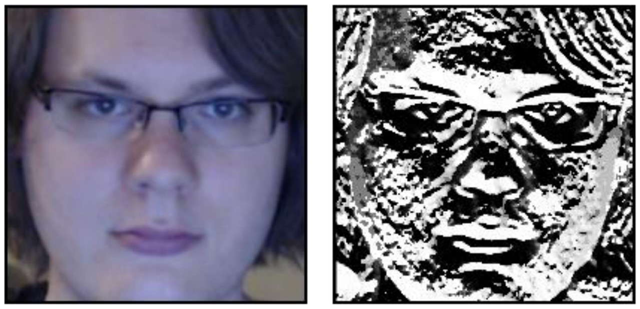
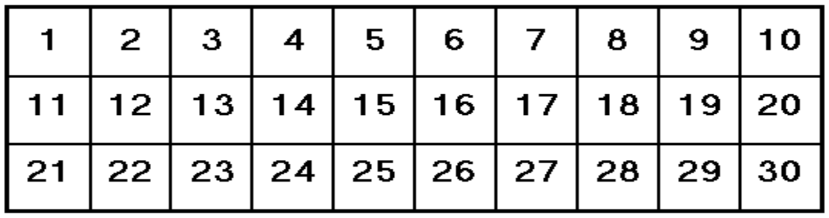


Figure 3.4: *The image on the left is a cropped face (detected using Haar-Cascade classifier) from thevideo frame. The image on the right is the LBP representation processed in real-time.*

Given this implementation, we were capable of acquiring test samples in approximately the same speed in which images were captured by the video camera. This was a critical requirement as our software application needs to process dozens of images a second in order to truly be real-time. Furthermore, the application also needs to feed data through a classifier, such as the SVM or neural network, which adds additional overhead. As such, it is important that all steps in the classification process run as efficiently as possible.

Exporting Data

In order to use the dataset with various classification algorithms, it is important to represent the data as feature vectors instead of images. In order to do this, pixel values are read from left to right starting at the top row. Each pixel value is appended to a 1 dimensional array. Fig. 5 provides an example.



1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30

Figure 3.5: *Array element order for a 10 x 3 image. Values are added from left-to-right, top-to-bottom, with each pixel representing a feature value.*

Given the transformations applied to the data, we can expect that each image will be in grayscale. With a grayscale image each pixel is represented by an integer from 0-255. The higher the value the darker the pixel, from 0 (black) to 255 (white). Each pixel value is appended to a one dimensional array which represents the entire sample image. Furthermore, each sample is contained within a single row in the data matrix. Given this matrix, we perform two methods to export the data, one for SVM (MATLAB) and another for the neural network (TensorFlow/numpy).

For the SVM implementation, the data is written to a plain-text file with a space delimiter. This allowed the data to be imported easily into MATLAB. However, the downfall of this method was data size. Since a plain-text representation requires more space than an integer byte representation, the data size was much larger than the source images. However, for smaller images, the resulting data size was manageable. For example, our dataset with an image size of 64x64 required roughly 400MB in space while stored as JPG files whereas the exported pain-text representation required 1.2GB in space.

The neural network on the other hand required the data in a different format. The matrix needed to be transposed and placed within a tensor data structure. This process required a significant amount of compute time and can take hours to complete. The new data structure was stored within a numpy array and exported via Spyder. This method was also wasteful in space, transforming the dataset to 3X the original size similar to the plain-text implementation.

4. Conventional approach

Kth Nearest Neighbor

The first method we tried to solve this classification problem is the kth nearest neighbor as it is one of the easier methods to implement.

Theory

The nearest neighbor method is to find a predefined number of training samples closest in distance to the new point and predict the label from these distances. It is a non-parametric method, and is found to be successful in classification situations where the decision boundary is very irregular.

The details of algorithm is listed below[7]:

$$p_n(x \mid \omega_i) = \frac{k_i / n_i}{V},$$

where $n =$ total number of samples in the training set

$n_i =$ number of samples in class i,

$k_i =$ number of neighbors in V in class ω_i,

$V =$ volume inclusive of k neighbors.

$$p(x) = \sum_{i=1}^{c} p(x \mid \omega_i) P(\omega_i) = \sum_{i=1}^{c} \frac{k_i / n_i}{V} \frac{n_i}{n} = \frac{\sum_{i=1}^{c} k_i}{nV} = \frac{k}{nV}.$$

Therefore,

$$g_i(x) = \frac{\frac{k_i / n_i}{V} \frac{n_i}{n}}{\frac{k}{nV}} = \frac{k_i}{k} = \text{fraction of } k\text{-nearest neighbors from class } i.$$

Application and Results

The testing is performed on 100k set of data each have 26 dimensions. We separated 10k set of data to use as testing group the rest are the training group. After the testing, we found out this classification runs relatively slow as it needs to perform a lot of square calculations of floats. This floating point operation is relatively slow compared to other types such as 8-bit integers.

The Kth nearest neighbor algorithm provided us with 21% accuracy which is not ideal. Given the poor performance, we attempted to use the data set with image size 64x64, resulting in 4096 dimensions. Unfortunately by doing this our run-time increased by 160 times which would take dozens of hours to complete and is therefore impractical for any real-time application.

5. SVM (Support vector machines) Approach

As an alternative to using kth nearest neighbor with 4096 dimensions, we decided to try the Support Vector Machine (SVM) approach as it is capable of processing high dimensional data.

Theory

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection. They are primarily a classier method that performs classification tasks by constructing hyperplanes in a multidimensional space that separates cases of different class labels. SVM supports both regression and classification tasks and can handle multiple continuous and categorical variables. SVMs maximize the margin (Winston terminology: the 'street') around the separating hyperplane. The decision function is fully specified by a (usually very small)subset of training samples, the support vectors. This becomes a Quadratic programming problem that is easy to solve by standard method.[6] The details of algorithm can be seen below:

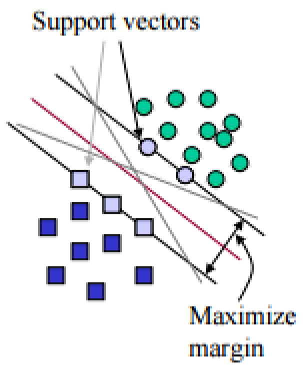


Figure5.1: SVM basic concept

Given training vectors $x_i \in \mathbb{R}^p$, i=1,..., n, and a vector $y \in \mathbb{R}^n$ ε-SVR solves the following primal problem:

$$\min_{w,b,\zeta,\zeta^*} \frac{1}{2}w^T w + C\sum_{i=1}^{n}(\zeta_i + \zeta_i^*)$$

$$\begin{aligned} \text{subject to } & y_i - w^T\phi(x_i) - b \leq \varepsilon + \zeta_i, \\ & w^T\phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*, \\ & \zeta_i, \zeta_i^* \geq 0, i = 1, ..., n \end{aligned}$$

Its dual is:

$$\min_{\alpha,\alpha^*} \frac{1}{2}(\alpha - \alpha^*)^T Q(\alpha - \alpha^*) + \varepsilon e^T(\alpha + \alpha^*) - y^T(\alpha - \alpha^*)$$

$$\begin{aligned} \text{subject to } & e^T(\alpha - \alpha^*) = 0 \\ & 0 \leq \alpha_i, \alpha_i^* \leq C, i = 1, ..., n \end{aligned}$$

where e is the vector of all ones, $C > 0$ is the upper bound, Q is an n by n positive semidefinite matrix, $Q_{ij} \equiv K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ.

The decision function is:

$$\sum_{i=1}^{n} (\alpha_i - \alpha_i^*) K(x_i, x) + \rho$$

These parameters can be accessed through the members dual coefficient which holds the difference, support vectors_ which holds the support vectors, and intercept which holds the independent term

Kernel Functions

$$K(\mathbf{X}_i, \mathbf{X}_j) = \begin{cases} \mathbf{X}_i \cdot \mathbf{X}_j & \text{Linear} \\ (\gamma \mathbf{X}_i \cdot \mathbf{X}_j + \mathrm{C})^d & \text{Polynomial} \\ \exp(-\gamma \,|\, \mathbf{X}_i - \mathbf{X}_j \,|^2) & \text{RBF} \\ \tanh(\gamma \mathbf{X}_i \cdot \mathbf{X}_j + \mathrm{C}) & \text{Sigmoid} \end{cases}$$

where $K(\mathbf{X}_i, \mathbf{X}_j) = \phi(\mathbf{X}_i) \bullet \phi(\mathbf{X}_j)$

Application and Results

The testing is performed on 100k samples in the data set, with each sample represented in 26 dimensions. We separated the samples into 90k for training and 10k for the test data.

The 26-dimensional data allowed us to finish the test in around 1.5 seconds. Since the SVM has a lot of parameters that we can set, we decided to use a nested for loop to go through all the possible combinations. After 11 hours of running, we found a general trend in the parameter settings. These trends are discussed below.

One of the trends that we found is the cost function has the most impact on the SVM performance. We found that no matter what other parameters we set, we got our best result at a cost parameter of 21.

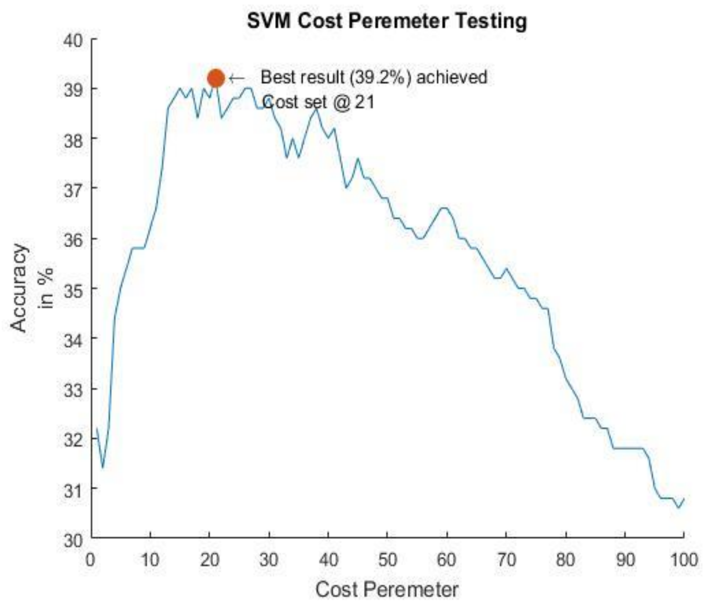


Figure5.2: SVM cost parameter testing

The other trend that we found is we received the best results using a linear SVM classifier

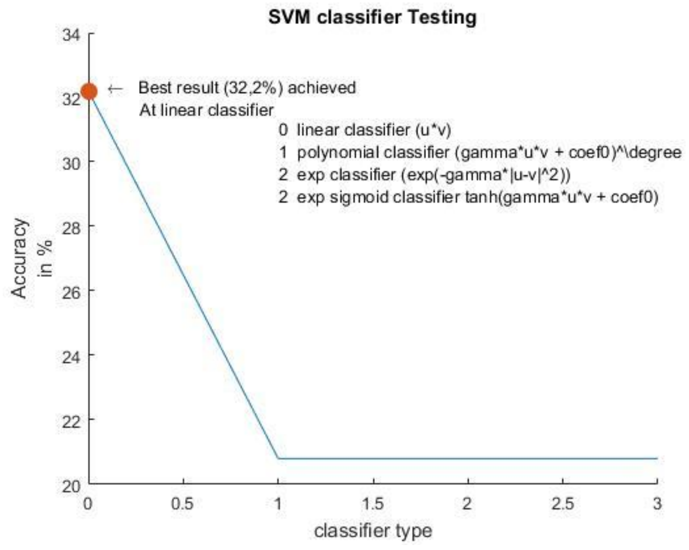


Figure5.3: SVM classifier testing

The best accuracy we can achieve on the 26-dimensional data is 39.2%.

After we found the best combination of settings we changed our data set to use the total 4096 dimensions from the LBP images. Calculation time with this data set was much longer but resulted in higher accuracy. The best accuracy we managed to get using this method was 86%.

As an additional test in dimension reduction, we implemented the PCA algorithm. However, the results were worse than the 26-dimensional performance and as such was not useful as our classifier. This makes sense as SVM can take advantage of data with higher dimensions. Even though PCA will reduce dimensions, this dimension reduction will hurt performance in comparison to the SVM algorithm.

6. CNN (Convolutional Neural Networks) Approach

CNN Theory

Convolutional Neural Networks are built with neurons that have weights and biases that can be update with each iteration. Each neuron receives some inputs, produces a dot product operation, and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to the class scores on the other. The last layer implements a loss function similar to other neural networks.

Convolutional Neural Network architectures usually deal with inputs such as images, which allow us to encode certain properties into the architecture. Forward propagation is more efficient to implement and vastly reduces the amount of parameters in the network due to the structure.

Convolutional Neural Networks benefit from the fact that the input consists of images, where the network can compress the image data in a more sensible way. Unlike regular neural networks, the layers of a Convolutional Network have neurons arranged in the following dimensions: width, height, and depth. The neurons in a layer will only be connected to a small part of the layer before it, whereas normally it is fully connected to the previous layer. The result of this architecture is that the full image will be transformed into a single vector of class scores, arranged along the depth dimension. Fig. 6.1 provides an example.

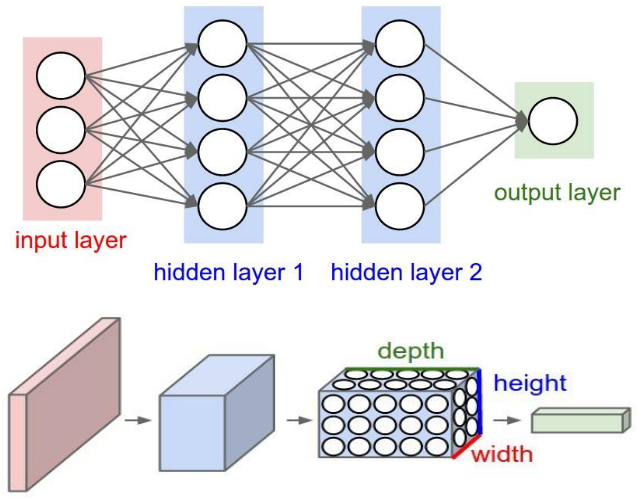


Figure 6.1: CNN Architecture

The top image in Fig. 6.1 shows a regular 3-layer Neural Network. The bottom image shows how the CNN builds the neurons in three dimensions. Every layer of the CNN transforms the 3D input volume to a 3D output volume after activation. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (in Red, Green, Blue channels).

As we described above, a simple CNN is combined by a group of layers, and every layer transmutes one volume of activations to another through a differentiable function. We use three main types of layers to build the following: Convolutional Layer, Pooling Layer, and a Fully-Connected Layer.

The Architecture we used can be found at [4] and is listed below:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.

- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.

- RELU layer will apply an elementwise activation function, such as the max(0,x)max(0,x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).

- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].

- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

By doing this, our CNN can change each pixel of the original image to the desired class score. Each layer has a unique function. For example, the CONV/FC layers perform transformations that are a function of both the activations of the input volume, and the parameters which are the weights and biases of the neurons. On the other hand, the RELU/POOL layers will implement a fixed function. [4]

Application of Convolutional Neural Networks

In order to build a good Convolutional Neural Networks. There are many things that we should choose.

A. choose input data size and pattern

First, we should choose the input data. Our raw data is 5,876*200*200*1, which means 5,876 samples, 200 in length,200 in width and 1 in channel. For the image sizes, we tried 200x200, 128x128 ,64x64 and 32x32. Due to performance limitations of our computers, we decided to use 32x32 as our sample image size since the smaller images are easier to process.

As we mentioned before, our original data is combined samples generated through applying noise. This combination results in a data set with over 90,000 samples. In theory, we should set the original data as the test data since these images will be the most similar to the samples we gather in real time. Fig. 6.2 – 6.5 shows how these data sets work differently in a Convolutional Neural Network.

1. Using the original data or the generated noisy data as the test data

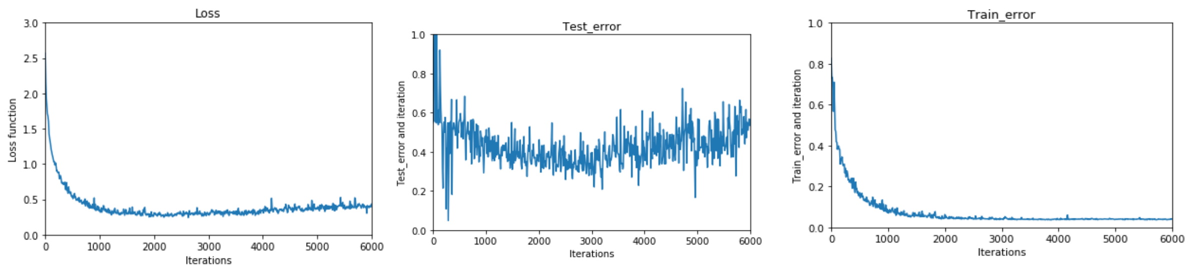


Figure 6.2: *Using generated data as test data loss, test error and train error*

2. Using the original data as the test data

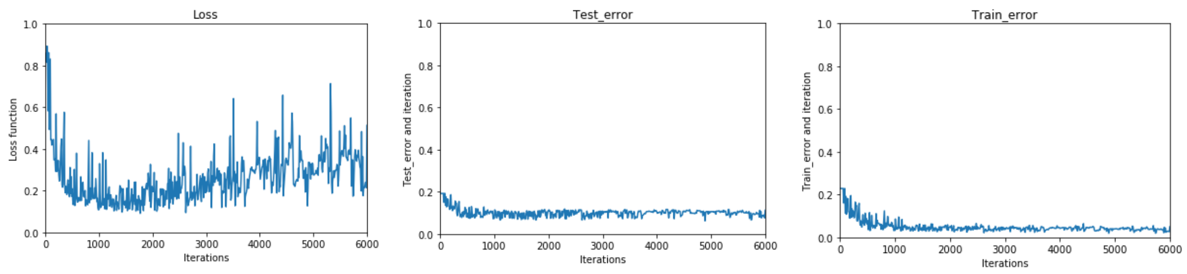


Figure 6.3: *Using original data as test data loss, test error and train error*

It's clear that when using original data system performance is better. But the performance of using the generated noisy data as the test data is really interesting. The test error approaches the minimum at very early iterations, then becomes larger dramatically. Given this finding, we decided to use the original data as our test data which makes the system more robust against noise.

3. Using LBP data or the original data as the test/training data

LBP Data

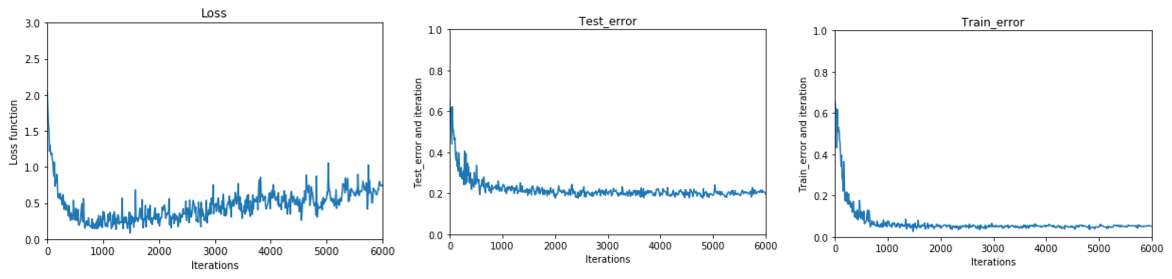


Figure 6.4: *Using LBP data as test data loss, test error and train error.*

Original Data

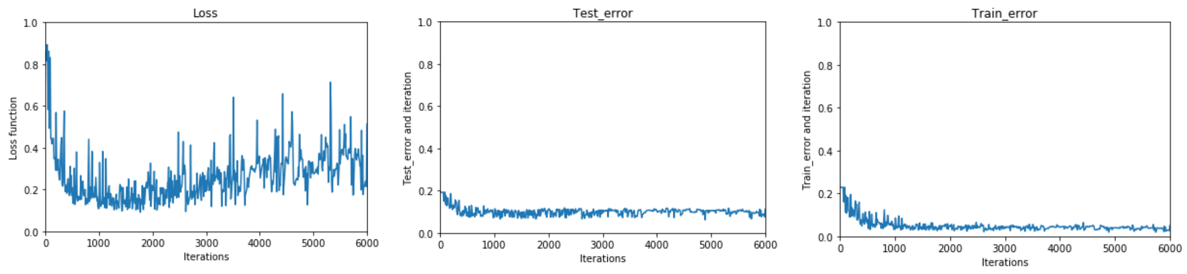


Figure 6.5: *Using original data as test data loss, test error and train error.*

Using the original data vs an LBP representation results in similar accuracy. Given that the accuracy results are similar, we decided to implement both of them in real-time.

B. choose number of layers for the Convolutional Neural Network

In choosing the number of layers, we chose the original data without LBP to test our model. For Neural Networks that contain only an input layer, hidden layer and output layer, the number of parameters can be estimated in the following way:

n_H determines complexity of decision boundary , Too many hidden nodes results in “overfitting” to the training data – poor generalization and loss of performance: see test data.

choose n_H such that:

Number of weights (w) = n/10, where n is the number of training patterns.

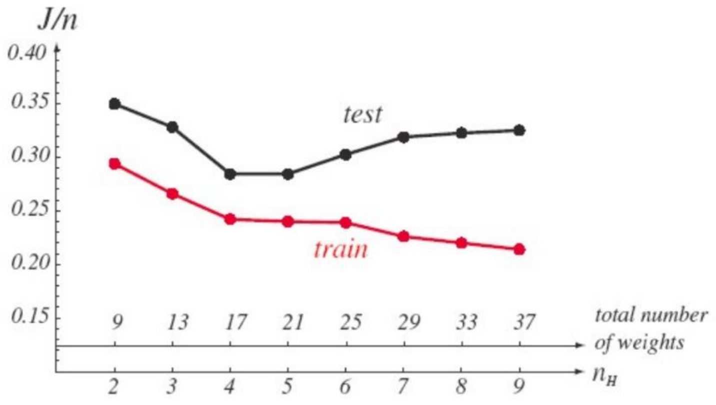


Figure 6.6: *Accuracy vs n_H, total number of weights [8]*

For Convolutional Neural Networks, in our experience, the more convolutional layers the better (within reason, as each convolutional layer reduces the number of input features to the fully connected layers), although after about two or three layers the accuracy gain becomes rather small so you need to decide whether your main focus is generalization accuracy or training time. That said, all image recognition tasks are different so the best method is to simply try incrementing the number of convolutional layers one at a time until you are satisfied by the result.

In Convolutional Neural Networks, the higher the dimensions of the input data, the larger the layers should be. For our raw image data of size 200x200, we need 4 layers to reduce the dimensions layer by layer. However, since we decided to use 32x32 image data, 3 layers are sufficient. We also tried 2 and 4 layers but 3 was the most efficient.

Model 1

Our model 1 of 3 layers show as below:

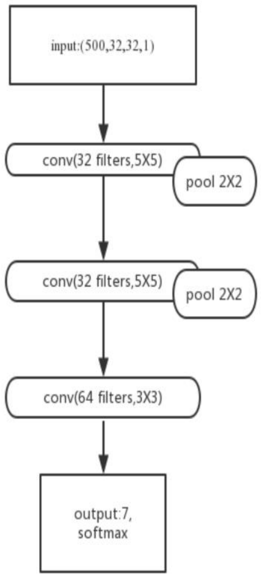


Figure 6.7: *CNN model 1*

The calculation of each layer show as below:

Input in batch is [500,32,32,1]

weight of convolutional layer1: [5,5,1,32] (total number: 5*5*3*32)

bias of convolutional layer1: [28,28,32]

hidden layer of convolutional layer1: [500,28,28,32]

hidden layer of convolutional layer after polling 1: [500,14,14,32]

weight of convolutional layer 2: [5, 5, 32, 32] (total number: 5*5*32*32)

bias of convolutional layer 2: [14,1432]

hidden layer of convolutional layer 2: [500,10,10,32]

hidden layer of convolutional layer after polling 2: [500,5,5,64]

weight of convolutional layer 3: [3, 3, 32, 64] (total number: 3*3*32*64)

bias of convolutional layer 3: [3,3,64]

hidden layer of convolutional layer 3: [500, 3,3,64]

hidden layer of convolutional layer3_flat: [500, 3*3*64]

Weight of fully connected layer 2: [3*3*64, 7] (total number: 3*3*64*7)

Bias of fully connected layer 2 = [7]

output layer: [500,7]

Model 2

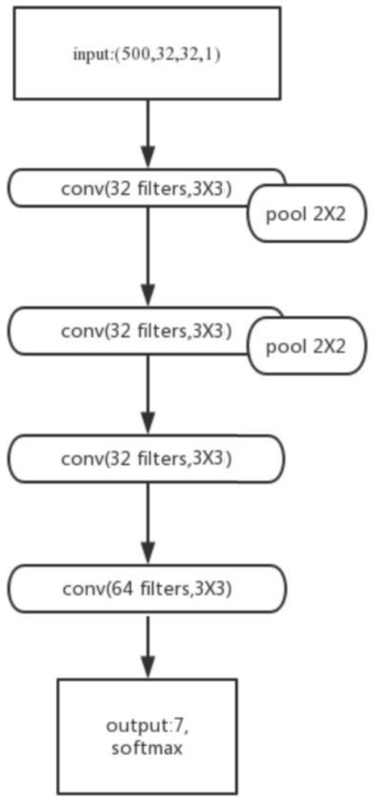


Figure 6.8: *CNN model 2*

The calculation of each layer show as below:

Input in batch is [500,32,32,1]

weight of convolutional layer1: [3,3,1,32] (total number: 3*3*3*32)

bias of convolutional layer1: [30,30,32]

hidden layer of convolutional layer1: [500,30,30,32]

hidden layer of convolutional layer after polling 1: [500,15,15,32]

weight of convolutional layer 2: [3, 3, 32, 32] (total number: 3*3*32*32)

bias of convolutional layer 2: [15,15,32] hidden layer of convolutional layer 2: [500,13,13,32]

hidden layer of convolutional layer after polling 2(with zero padding): [500,7,7,64]

weight of convolutional layer 3: [3, 3, 32, 32] (total number: 3*3*32*32)

bias of convolutional layer 3: [5,5,32]

hidden layer of convolutional layer 3: [500, 5,5,64]

weight of convolutional layer 4: [3, 3, 32, 64] (total number: 3*3*32*64)

bias of convolutional layer 2: [3,3,32]

hidden layer of convolutional layer 2: [500,3,3,32]

hidden layer of convolutional layer3_flat: [500, 3*3*64]

Weight of fully connected layer 2: [3*3*64, 7] (total number: 3*3*64*7)

Bias of fully connected layer 2 = [7]

output layer: [500,7]

Model 3

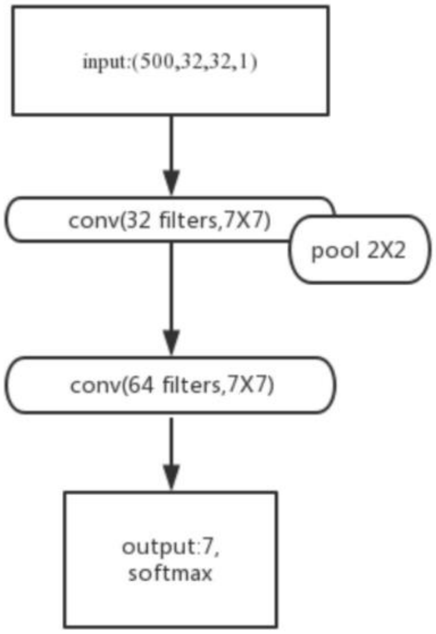


Figure 6.9: *CNN model 3*

The calculation of each layer show as below:

Input in batch is [500,32,32,1]

weight of convolutional layer1: [7,7,1,32] (total number: 7*7*1*32)

bias of convolutional layer1: [26,26,32]

hidden layer of convolutional layer1: [500,26,26,32]

hidden layer of convolutional layer after polling 1: [500,13,13,32]

weight of convolutional layer 2: [7, 7, 32, 64] (total number: 7*7*32*64)

bias of convolutional layer 2: [7,7,64]

hidden layer of convolutional layer 2: [500, 7,7,64]

hidden layer of convolutional layer2_flat: [500, 7*7*64]

Weight of fully connected layer 2: [7*7*64, 7] (total number: 7*7*64*7)

Bias of fully connected layer 2 = [7]

output layer: [500,7]

Accuracy Results

Accuracy of test data	LBP data (without drop out)	Original (without drop out)	LBP (with dropout)	Original (with dropout)
model1	0.8763	0.8831	0.9157	0.9209
model2	0.7984	0.7754	0.8854	0.8753
model3	0.8061	0.8127	0.8932	0.8974

Table 6.1 Accuracy Result between models

From the label it's clear that model 1 has the best performance, and in theory 3 layers is enough for a input with (batch_size, 32, 32, 1). The performance in model 2 was not optimal, likely due to a lack of enough layers in the CNN.

C. Parameters Tuning

After we find the best performance model, we tuned the parameters of the system to achieve better accuracy. The parameters we can modify are the following: Learning rate, learning rate decay, and dropout probability.

First, learning rate determines the performance of system. A high learning rate leads to overshooting, while a learning rate that is too small may result in the system getting stuck in local minima. For this system, we choose a learning rate of 0.001.

Second, we should choose the decay of the learning rate so that learning rate will decrease after each iteration. At first, the learning rate is large enough to avoid getting stuck in local minima, and after several iterations it will become small enough to avoid overshooting.

Third, we should choose the dropout probability. Dropout is a technique that addresses both overfitting and getting stuck in local minima. It prevents overfitting by providing a way of approximately combining exponentially many different neural network architectures efficiently. The term “dropout” refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections. The choice of which units to drop is random.

CNN result on LBP data:

Accuracy of test data	Learning rate= 0.001; decay=0.9	Learning rate= 0.001; decay=0.8	Learning rate= 0.001; decay=0.7	Learning rate= 0.002; decay=0.9	Learning rate= 0.002; decay=0.8	Learning rate= 0.002; decay=0.7
dropout =0.4	0.9063	0.9091	0.9143	0.9124	0.9216	0.9143
dropout =0.5	0.8914	0.9274	0.9327	0.9278	0.9214	0.9221
dropout =0.6	0.9061	0.9127	0.9232	0.9261	0.9257	0.9256

Table 6.2: *CNN result on LBP data*

CNN result on original data:

Accuracy of test data	Learning rate= 0.001; decay=0.9	Learning rate= 0.001; decay=0.8	Learning rate= 0.001; decay=0.7	Learning rate= 0.002; decay=0.9	Learning rate= 0.002; decay=0.8	Learning rate= 0.002; decay=0.7
dropout =0.4	0.9170	0.9154	0.9380	0.9267	0.9254	0.9267
dropout =0.5	0.9019	0.9249	0.9376	0.9127	0.9298	0.9237
dropout =0.6	0.9164	0.9167	0.9287	0.9298	0.9268	0.9224

Table 6.3: *CNN result on original data*

For both the original and LBP data set, the system achieved the best performance with the following parameters:

Learning Rate=0.001

Decay=0.7

Dropout=0.4 or 0.5.

7. Real-Time Implementation

In order to achieve real-time classifications, we trained the neural network to obtain a working model and then sent test samples through the model to be classified. In order to achieve test samples we used OpenCV which was able to collect images from an attached webcam. Each raw frame send to a Haar cascade classifier in order to detect if any faces are present. If so, the faces are organized along their center position on the x-axis. Each face is cropped out of the original frame and converted to grayscale using OpenCV. This image is then processed by Scikit-image in order to obtain the local binary pattern. The image is saved to a JPG file temporary which is then read in by the neural network as a test sample.

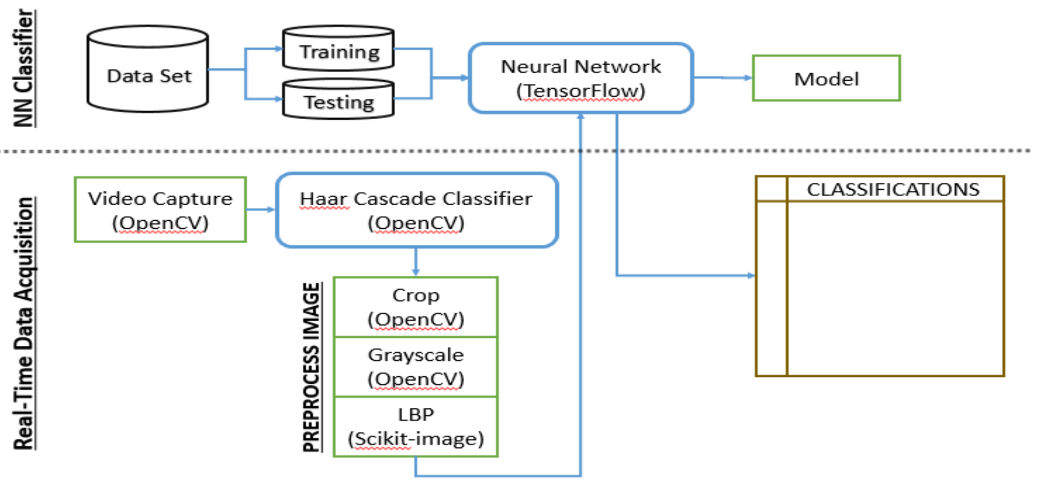
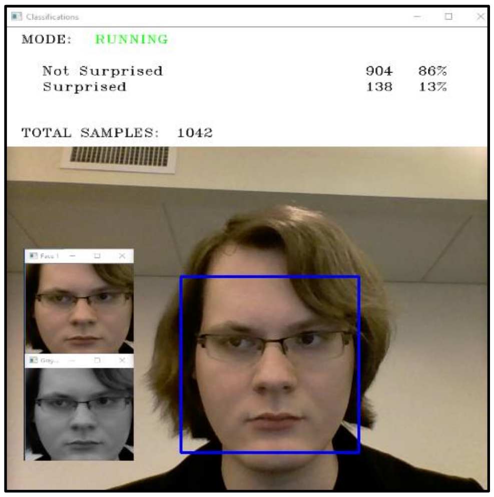


Figure 7.1: *Data flow for the neural network classifier and real-time application.*

Given the neural network has successfully built a model from the training and test data, the real-time data acquisition process can use the model to determine a classification. Each class has a counter to record the classification frequency in regards to the total frames captured. These counters along with their percentage of the overall frames is displayed on the screen to the end-user. This provides a clear visual way to determine how accurate the classification model is when tested in real-time. During the data acquisition process, the user is capable of the following commands through the terminal: standby mode, classification mode, and resetting classification statistics.

Our initial test for real-time classification ended in poor performance when using 7 emotion classes. The results would vary drastically in regards to the individual being tested and the parameters used in building the model. Considering that the neural network was able to achieve 93.76% accuracy on grayscale samples and 93.27% on LBP sample, we suspect that the images collected in real-time have too many unique characteristics in order to fit into the 7-class model. Instead, we simplified our model to 2 classes: Surprised and Not Surprised. The results of these tests can be seen below:

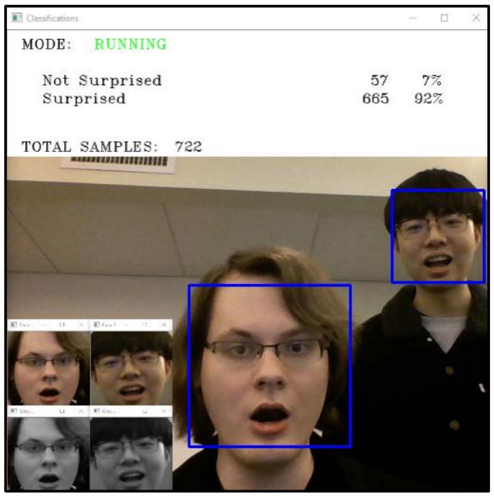


```
MODE:   RUNNING

  Not Surprised                     904   86%
  Surprised                         138   13%

TOTAL SAMPLES:  1042
```

Figure 7.2: <u>**Real-Time Classification Trial 1**</u>
Actual Classification: Not Surprised
Total Samples: 1042
Error Rate: 13.24% Error Rate.



```
MODE:   RUNNING

  Not Surprised                      57    7%
  Surprised                         665   92%

TOTAL SAMPLES:  722
```

Figure 7.3: <u>**Real-Time Classification Trial 2**</u>
Actual Classification: Surprised
Total Samples: 722
Error Rate: 7.89% Error Rate.

Both tests resulted in strong performance, both with a single face and multiple faces sharing the same expression. These results verify the correctness of our learning algorithm and show an overall capacity to learn facial expressions. In order to utilize all 7 emotion classes we believe further fine tuning would need to be performed in order to either improve the robustness of the model or make the real-time samples closer resemble the original dataset.

8. Conclusion

In our experiments, we found that Kth nearest neighbor did not provide very good results. Also, the calculation time for this method is too slow in order to implement in real-time.

The SVM algorithm, however, was good at separating high dimensional data and achieved high accuracy in our facial expression recognition when using 4096 dimensional LBP samples. We could achieve an accuracy of 82% after we adjusted all the parameters. The training time is just 5 minutes for an 1GB data set. Unfortunately the SVM implementation performed too slowly in order to be practical in the real-time application.

CNN algorithm ended up as a powerful classifier which could deal with both large data sets and high dimensionality, as well as provide high accuracy for classifications. We were able to achieve 93.76% accuracy with our data set. Training on a CNN cost nearly half an hour, however, once the model is built, test samples can be classified with very little overhead. This allowed our real-time application to process dozens of samples per second.

9. Future work

In our testing we found that the SVM algorithm had a better result on data sets with high dimensionality. However, as stated in the introduction, research has found that by applying Gabor wavelets will improve accuracy and reduce dimensionality [1], which should in turn make our classification algorithm run even faster in real-time.

One drawback to the approach in this paper is the data set we use. We believe that if we built our own data set using only frames that clearly express a given emotion, we could achieve better results. Furthermore, using the same camera for both the model training and real-time data acquisition should result in better accuracy.

References

[1] Zhang, S., Zhao, X., & Lei, B. (2012). Robust facial expression recognition via compressive sensing. *Sensors*, *12*(3), 3747-3761.

[2] Levi, G., & Hassner, T. (2015, November). Emotion recognition in the wild via convolutional neural networks and mapped binary patterns. In Proceedings of the 2015 ACM on International Conference on Multimodal Interaction (pp. 503-510). ACM.

[3] Vupputuri, A., & Meher, S. (2015, April). Facial Expression recognition using Local Binary Patterns and Kullback Leibler divergence. In Communications and Signal Processing (ICCSP), 2015 International Conference on (pp. 0349-0353). IEEE.

[4] K. (n.d.). CS231n: Convolutional Neural Networks for Visual Recognition. Retrieved May 02, 2017, from http://cs231n.github.io/convolutional-networks/

[5] T. Kiran and T. Kushal, "Facial expression classification using Support Vector Machine based on bidirectional Local Binary Pattern Histogram feature descriptor," 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Shanghai, 2016, pp. 115-120. doi: 10.1109/SNPD.2016.7515888

[6] Smola, A. J., & Schölkopf, B. (2004). A tutorial on support vector regression. *Statistics and computing*, *14*(3), 199-222.

[7] Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory*, *13*(1), 21-27.

[8] Duda, R. O., Hart, P. E., & Stork, D. G. (n.d.). Pattern classification.