Anoma Research Topics | TECHNICAL REPORT

Report of Research

Fatemeh Afshari

* *E-Mail:*

Abstract

In this dissertation, The publishable part of my research is shared, all the projects are open and will improve. I will be grateful for any comments on my work.

Keywords: Neural Networks ; Delaunay Triangulation ; PCA ; machine learning ; Q-learning ;

Contents

1. Introduction

This dissertation is devoted to reporting some of my Pyhton projects that I have been working on lately. The main scope of this is about my research into face expression recognition.
The first two sections feature two elementary projects in the area of neural

networks. The first, devises a simple neural network to predict the price of cars in a dataset. The second section designs more complicated neural networks to classify cell images.
The rest of the report focuses more on the research into face expression recognition problems. Sections 3 and 4 implement two articles that proposed architectures for neural networks for face expression recognition. Section 5 encounters the problem of face recognition with integration of machine learning and Delaunay triangulation methods. Finally, the 6th section tries to solve the face recognition problem with the integration of two methods of PCA and Delaunay Triangulation. I am researching face expression recognition and all my projects in this area are open ones that I am still working on.
Furthermore, I have published an article at the beginning of this year with the subject that is breifly discussed in the last section.

2. Predict Car Price

In this project the price of second-hand cars is predicted based on their features. The data is loaded from kaggle dataset. This dataset consists of 8 columns that describe different features of 'years', 'km' , 'rating', 'condition', 'economy', 'top speed', 'hp' horsepower, 'torque', 'current price' and corresponding price. Most of the column's values have the standard distribution. The prediction of the price is done using a trained model. This model is trained by the neural networks algorithm.

Neural network algorithms are inspired by the human neuron system. This algorithm establishes layers of neurons that are connected to each other. Each connection between the neurons has a weight. As is shown in figure 1. In the first layer, there are some input values that activate the corresponding neurons and these activated neurons will activate the connected neurons according to the weight of their connection. In each neuron, the sum of the weights from the previous layer of neurons and a bias value are summed. Finally, an activation function is applied and decides whether to activate the neuron or not.

The added libraries that are used in this project are listed in figure 2. Pandas is a library used for working with datasets, especially csv files, which is the type of our dataset in this project. Pandas provide functions for analyzing, cleaning, exploring, and manipulating data. The other library that is used is Seaborn, which is a Python data visualization library based on matplotlib, which is used for creating static, animated, and interactive visualizations. Seaborn provides a high-level interface for drawing attractive and informative statistical graphics. The numpy library is added to support large, multidimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The most important li-

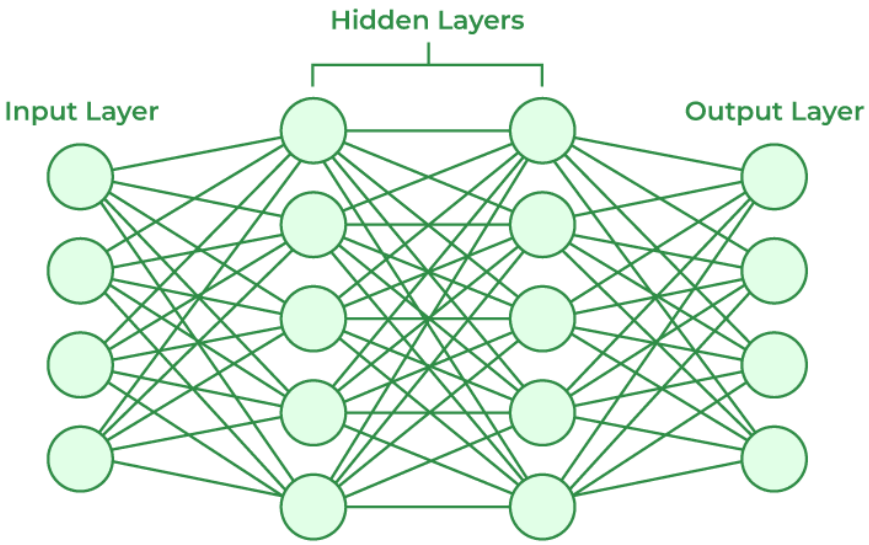


Figure 1. Input layer, two hidden layers, output layer

brary that I use on this project is Tensorflow, which features keras library. TensorFlow is a library for machine learning and artificial intelligence. It has a particular focus on training and inference of **deep neural networks**. The keras is used to import layers (Normalization, Dense and InputLayer, the loss function of MeanAbsoluteError and optimizer Adam.

```
import pandas as pd
import seaborn as sns
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Normalization, Dense, InputLayer
from tensorflow.keras.losses import MeanAbsoluteError
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
])
```

Figure 2

To visualize the data and dependencies between the columns, the seaborn library is used. The resulting diagram is shown in figure 3. To show the relation of the columns, the method seaborn.pairplot() is used, the type of the diagram is specified 'kde', which stands for kernel density estimate. The diagonal shows the relationship of the data column with itself. It can be derived that a parameter of km (the number of kilometers ridden by the car) has a correlation with the price.

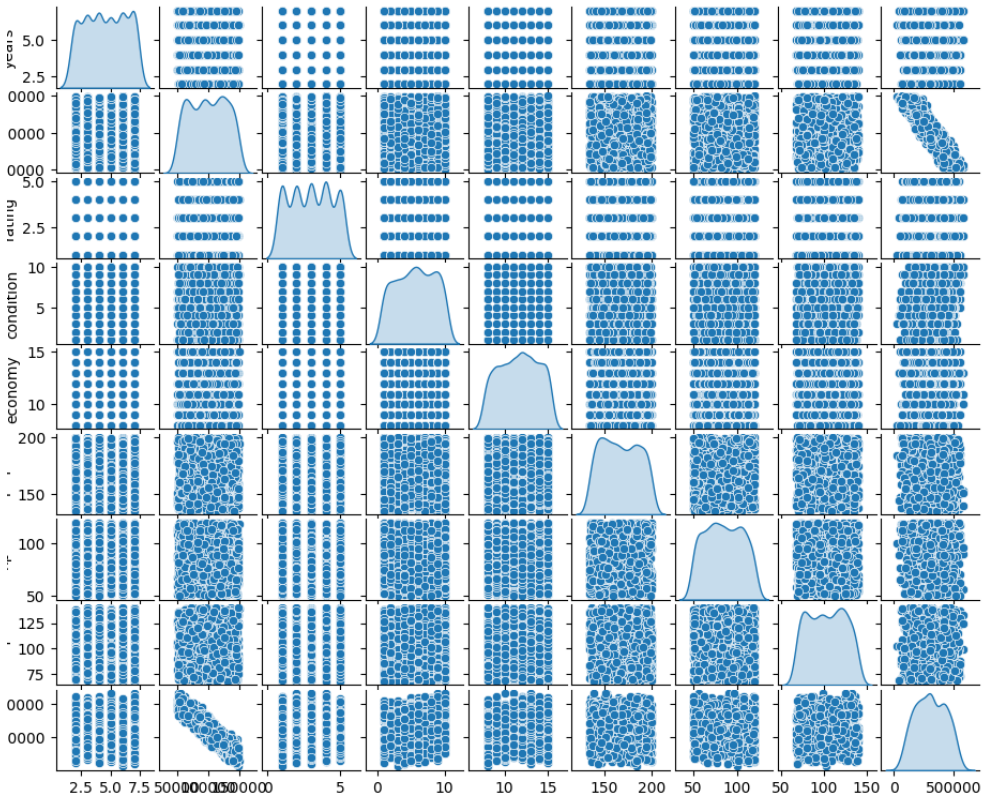


Figure 3. The result of the method seaborn.pairplot() with the kind of the diagram specified as 'kde' that stands for kernel density estimate

Next, the data is converted to tensor, normalized, and shuffled. Then, with respect to ratios of 8, 1, 1 which correspond to train, validation and test data, the dataset is split. When the data is prepared. A sequential model is generated to be trained. The sequential model is a tensorflow model that is appropriate for the plain and simple structure of layers and has one input and output tensor. The code for the sequential model is shown in 4. The sequential model is established with layers of Inputlayer, normalizer, Dense, and output layers. Inputlayer gets the input in the shape of (8,), the normalizer is a Normalization object with mean and standard deviation adapted to the train data. The Dense or fully connected layer is the layer in which all the neurons are connected to the inputs and outputs. In the model, three Dense layers are built with 128 neurons in each and an activation function of relu that is short for the Rectified Linear Unit and is shown in figure 5. The activation function decides whether a neuron is activated or not. Finally, a Dense layer with one neuron for output is added.

```
normalizer = Normalization()
normalizer.adapt(X_train)
model = tf.keras.Sequential([
                InputLayer(input_shape = (8,)),
                normalizer,
                Dense(128, activation = "relu"),
                Dense(128, activation = "relu"),
                Dense(128, activation = "relu"),
                Dense(1),
])
```

Figure 4

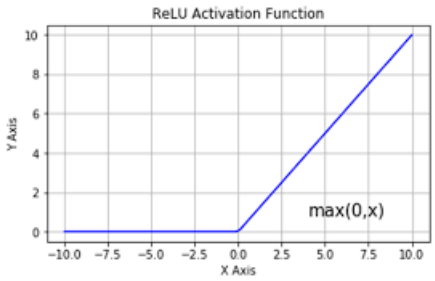


Figure 5. the Rectified Linear Unit for activation function in a neuron

Afterward, the model is compiled and fitted(figure 6). The compilation is done with the optimizer Adam, a learning rate of 0.1 and loss is computed by computing the Mean absolute error. Then, the training is done over 100 epochs.

```
model.compile(optimizer = Adam(learning_rate = 0.1),
              loss = MeanAbsoluteError())
history = model.fit(X_train, y_train, validation_data=(X_val, y_val),
    epochs = 100, verbose = 1)
```

Figure 6

In the figure 7, the evolution of the loss function on the train and validation data is shown. The loss in both sets is decreasing and reaching a plateau.

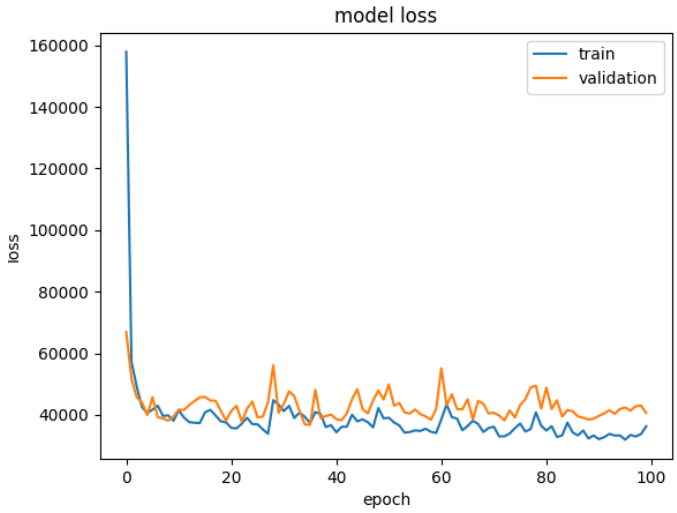


Figure 7

In the figure 8, the predicted values are drawn to compare with the actual value.

The code is available in github/fatemeafshar.

3. Malaria Detection

In this section, the details of the malaria detection project using cell images are expressed. The data consists of two classes of healthy and infected. The problem is to learn a model that can classify the new images as healthy or infected. The data is taken from kaggle.

First, a function is implemented for reading the images from directories. This code uses the function that is shown in figure 9 which is available in tensorflow version 2.17.0. This function has many great options to specify the data preprocesses, such as colormode that specifies how many channels the image is read. These features make it easy for this function to be utilized on many other projects. Similar to the previous section, the data is split into

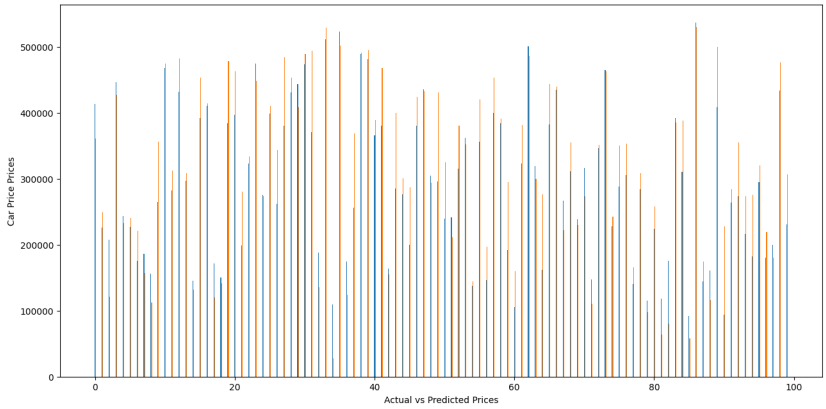


Figure 8. The compare of predicted value by the model and the actual value

train, validation and test sets.

```
train_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  label_mode = "categorical",#int
  color_mode = color_mode,
  image_size=(image_width, image_height),
  batch_size=batch_size)
class_names = train_ds.class_names
```

Figure 9

To solve this problem, a neural network has been established. Neural networks are discussed shortly in the 3. In the previous section, a sequential model using Dense or fully connected layers was built. In this section, a new structure of layers is discussed which is very common and useful for image data. The new structure is named convolutional neural network for its method of convolving the values. A convolution layer consists of filters or kernels. These kernels are 2D filters with trainable values that convolve into the data. This structure is immeasurably useful for image data, because features are within the little chunks of the photo which the convolutional layer is capable of extracting. In other words, image data (that is stored in pixels) because of its 2D nature, possesses features like edges, shapes etc. which can not be extracted from a set of unordered data, but it has to be extracted from related neighboring pixels. Therefore, the convolutional layer

that has a filter to consider these data orderly is amazingly sufficient for image data.

Another type of neural network that is used for images and is often implemented with convolutional layers is max pooling. Max pooling takes filter size and a stride value to apply over the data. The filter is applied to each block with the stride size steps. At each step, a maximum value is chosen to shape the new image. In this new image, only the most active pixels (pixels with maximum value) are brought. This process helps to decrease the size of data. This is especially useful in the case of image data that have large sizes. Convolution neural networks also reduce the size of the data during their operation, since they can not be operated on the edges properly. Although this reduces the size, it looses information on the edges. Tensorflow has resolved this loss of information on the edges by zero padding, which is basically adding zero values on the outer border of the image, to keep the information. However, zero padding keeps the size of the image high. To resolve this, a max pooling layer is added to keep the most activated values, without loss of information on the edges.

To solve this problem, the leNet model is utilized to classify the cell data. The overall leNet model is shown in figure 10 and the code generated to establish this architecture is shown in figure 11. The configuration of values is shown in the table1.

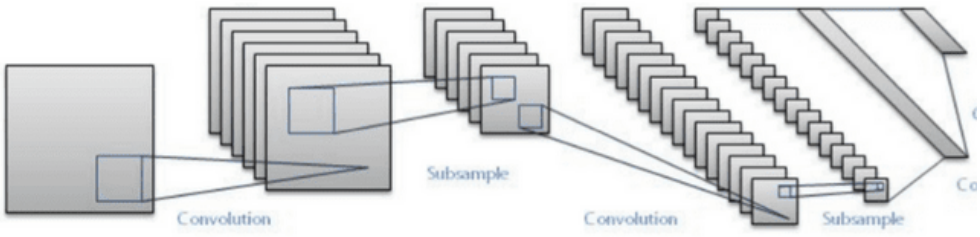


Figure 10

Table 1. A table showcasing the usage of the tabular environment.

Configurations	Values
LEARNING-RATE	0.001
N-EPOCHS	1
BATCH-SIZE	128
DROPOUT-RATE	0
N-FILTERS	6
KERNEL-SIZE	3
N-STRIDES	1
POOL-SIZE	2
N-DENSE-1	100
N-DENSE-2	10

```
    lenet_model = tf.keras.Sequential([
    InputLayer(input_shape=(IM_SIZE, IM_SIZE, 3)),

    Conv2D(filters=N_FILTERS, kernel_size=KERNEL_SIZE, strides=N_STRIDES,
           padding='valid',activation='relu',
           kernel_regularizer=L2(REGULARIZATION_RATE)),
    BatchNormalization(),
    MaxPool2D(pool_size=POOL_SIZE, strides=N_STRIDES * 2),
    Dropout(rate=DROPOUT_RATE),

    Conv2D(filters=N_FILTERS * 2 + 4, kernel_size=KERNEL_SIZE,
           strides=N_STRIDES, padding='valid',
          activation='relu', kernel_regularizer=L2(REGULARIZATION_RATE)),
    BatchNormalization(),
    MaxPool2D(pool_size=POOL_SIZE, strides=N_STRIDES * 2),

    Flatten(),

    Dense(CONFIGURATION['N_DENSE_1'], activation="relu",
           kernel_regularizer=L2(REGULARIZATION_RATE)),
    BatchNormalization(),
    Dropout(rate=DROPOUT_RATE),

    Dense(CONFIGURATION['N_DENSE_2'], activation="relu",
           kernel_regularizer=L2(REGULARIZATION_RATE)),
    BatchNormalization(),

    Dense(2, activation="sigmoid"),

])
```

Figure 11

The model is compiled and trained with the optimizer (Adam) and the loss function of binary cross entropy.
This problem is resolved by neural networks, but three different approaches are implemented. The first, similar to the last section, establishes a sequential model. The second one uses the functional classes and the third one creates the model from scratch using subclassing. Two other methods are used because they provide more flexibility and control over the network. These two options let us create networks that have different and more complex structures.
The functional API can handle models with non-linear topology, shared layers, and even multiple inputs or outputs by giving control over inputs and outputs of the layers. In this project, a feature extractor was devised that is used for leNet model. In the figure 12 the codes to create a feature extractor is shown. In the figure 13 the established leNet model using feature extractor is shown.

```
func_input = Input(shape = (IM_SIZE, IM_SIZE, 3), name = "Input␣Image")
x = Conv2D(filters = 6, kernel_size = 3, strides=1,
       padding='valid', activation = 'relu')(func_input)
x = BatchNormalization()(x)
x = MaxPool2D (pool_size = 2, strides= 2)(x)
x = Conv2D(filters = 16, kernel_size = 3, strides=1,
       padding='valid', activation = 'relu')(x)
x = BatchNormalization()(x)
output = MaxPool2D (pool_size = 2, strides= 2)(x)

feature_extractor = tf.keras.models.Model(func_input,
          output, name = "Feature_Extractor")
```

Figure 12

In addition to functional API, there is another way to establish neural networks. Subclassing provides more freedom to build custom layers. In the subclssing, a class is defined. This class inherits from the keras Model, which provides many features to build custom layers like train and test. In the figure **??**, the leNet model for malaria detection is implemented using subclassing. In the init function (constructor), all the needed layers are defined, and in the call function, the structure is implemented, which can be done by any logic programming that provides the freedom and flexibility to build complex custom neural networks.

The results of the trained model for functional API and sequential model after 10 epochs, with the loss function of binary cross entropy and learning rate of 0.001, are reported in table2. The evolution of loss value on train and validation data using sequential model, functional API and subclassing are shown in figures 15, 16 and 17. As is observed, the validation loss at the

```
func_input = Input(shape = (IM_SIZE, IM_SIZE, 3), name = "Input␣Image")
x = feature_extractor(func_input)
x = Flatten()(x)
x = Dense(100, activation = "relu")(x)
x = BatchNormalization()(x)
x = Dense(10, activation = "relu")(x)
x = BatchNormalization()(x)
func_output = Dense(2, activation = "sigmoid")(x)

lenet_model_func = tf.keras.models.Model(func_input,
          func_output, name = "Lenet_Model")
lenet_model_func.summary()
```

Figure 13

Table 2. A table showcasing the usage of the tabular environment.

Results	Sequential Model	Functional API
accuracy over train data	0.9873	0.9888
loss over train data	0.0382	0.0390
accuracy over validation data	0.9419	0.9485
loss over validation data	0.1940	0.2005
accuracy over test data	0.9600	0.9670
loss over test data	0.1288	0.1157

beginning is high but after some epochs it decreases and reaches a plateau. To prevent overfitting, 10 epochs have been chosen.

The code is available in github/fatemeafshar.

4. Face Expression Recognition using Neural Networks 1

Face expression recognition aims to capture the emotion of a face, which can be useful for numerous applications. There have been many algorithms devised and proposed to solve the face expression recognition problem. The most common one that has been very popular in recent years is neural networks. Neural networks are very suitable because of their ability to dig deep and extract features and especially the convolutional neural networks that are used in particular for image data.

The dataset used for this project is taken from kaggle. This dataset consists of seven different emotions: angry, disgust, fear, happy, neutral, sad, and surprise. These data are also read using the directory reading function that was explained in section 3.

As was explained in the previous section 3, the convolutional neural network is a feed-forward network that uses filters(kernels) to extract the features of the pixels of the data that are neighboring each other. As was mentioned, it is very appropriate for image data, since pixels are related to the neighboring

```
class LenetModel(tf.keras.models.Model):
  def __init__(self):
    super(LenetModel, self).__init__()

    self.feature_extractor = FeatureExtractor(8, 3, 1, "valid", "relu", 2)

    self.flatten = Flatten()

    self.dense_1 = Dense(100, activation="relu")
    self.batch_1 = BatchNormalization()

    self.dense_2 = Dense(10, activation="relu")
    self.batch_2 = BatchNormalization()

    self.dense_3 = Dense(2, activation="sigmoid")

  def call(self, x):
    x = self.feature_extractor(x)
    x = self.flatten(x)
    x = self.dense_1(x)
    x = self.batch_1(x)
    x = self.dense_2(x)
    x = self.batch_2(x)
    x = self.dense_3(x)

    return x


lenet_sub_classed = LenetModel()
lenet_sub_classed(tf.zeros([2, 50, 50, 3]))
lenet_sub_classed.summary()
```

Figure 14

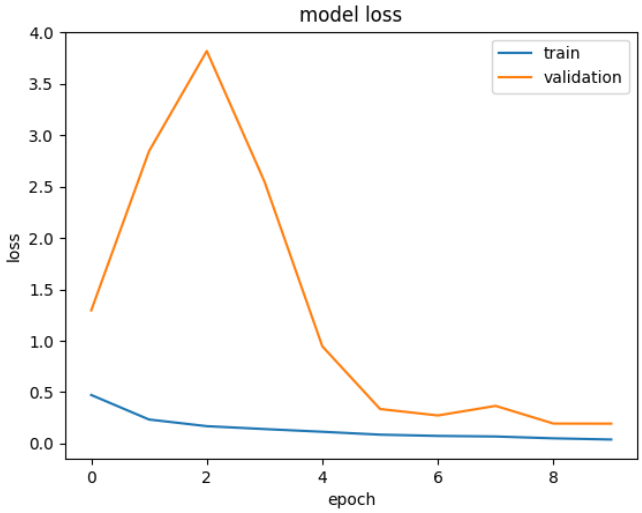


Figure 15

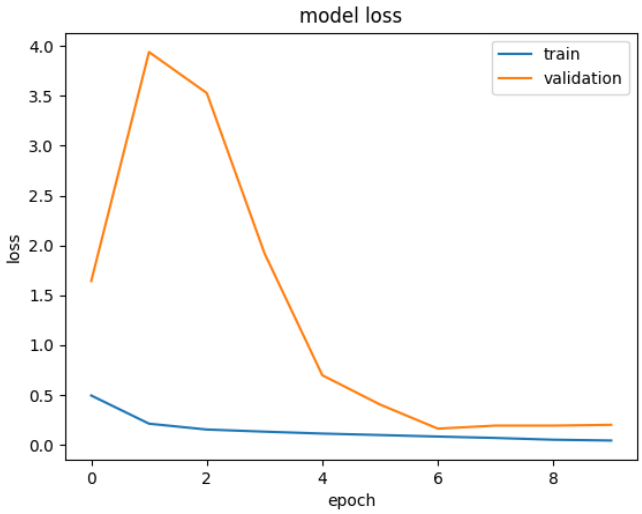


Figure 16

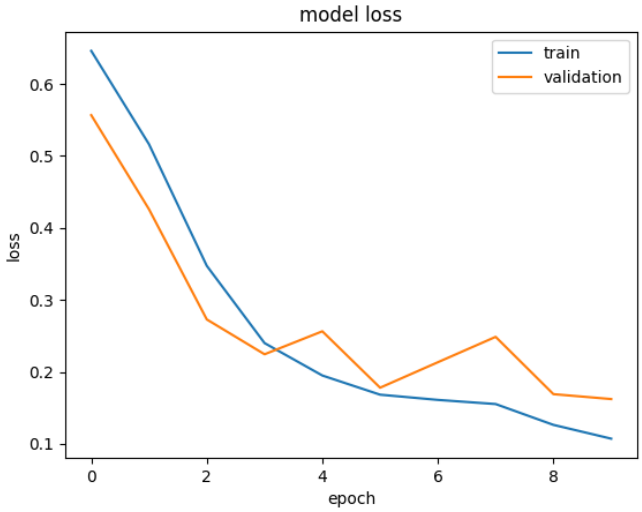


Figure 17

pixels, and the features of an image, such as edges, can be extracted by considering these pixels together.

Neural networks are a great tool to solve such a complex problem. There are a myriad of architectures and methods to devise neural networks for face expressions. In this section, [JRD20] is implemented. [JRD20] architecture has two data paths of layers, and at the end, it concatenates these two paths. At the final step, a dense layer is implemented as output for the classification. The architecture of the program is shown in the figure ??.

??. To implement architecture ?? that has a mild architecture because of the

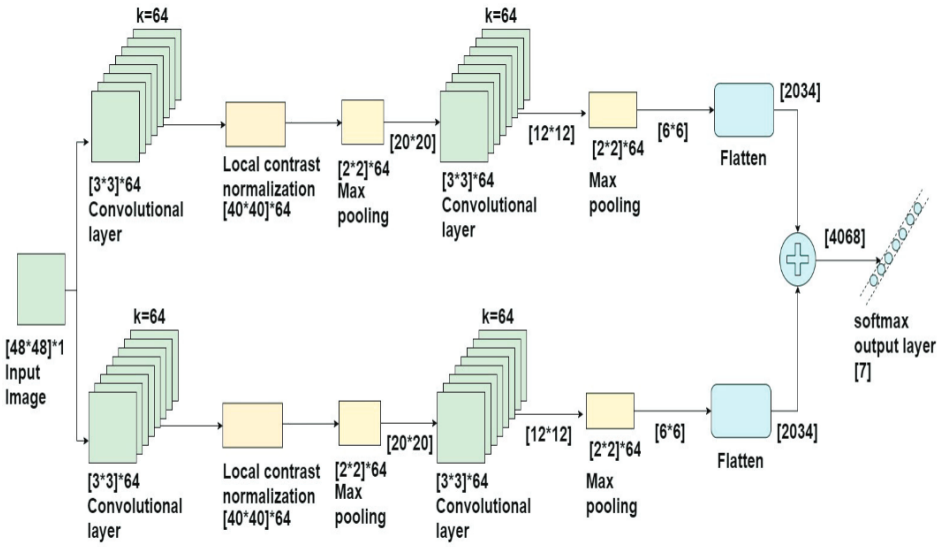


Figure 18. Two identical data paths that concatenate at the end

concatenation of paths, functional API is used. Functional API provides us with great means to establish more complex architectures, such as concatenated layers that have more than one input. The implemented code to build the model is shown in figure 19. The convolution layers are implemented with the padding value of the "same", which means that the size of the image remains the same by zero padding, which is explained in the 3. The normalization contrast is not implemented in the project(I am still working on this. This normalization is a filter based normalization that works according to the neighboring values).

```
_input = Input((CONFIGURATION['IM_SIZE'],CONFIGURATION['IM_SIZE'],1))

x = Conv2D(filters=64, kernel_size=(3,3), activation="relu")(_input)
# local contrast normalization
x = MaxPooling2D((2, 2))(x)
x = Conv2D(filters=64, kernel_size=(3,3), activation="relu")(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)
x = Flatten()(x)


y = Conv2D(filters=64, kernel_size=(3,3), activation="relu")(_input)
# local contrast normalization
y = MaxPooling2D((2, 2))(y)
y = Conv2D(filters=64, kernel_size=(3,3), activation="relu")(y)
y = BatchNormalization()(y)
y = MaxPooling2D((2, 2))(y)
y = Flatten()(y)

concatinated = Concatenate(axis=1)([x, y])
dropout_ = Dropout(0.2, noise_shape=None)(concatinated)
output = Dense(7, activation="softmax")(dropout_)
model = tf.keras.models.Model(inputs=_input, outputs=output)
```

Figure 19

At the end, a dropout layer with the rate of 0.2 is used. This means that some of the trained neurons are dropped to prevent overfitting of the model over the train dataset. All the activation functions are set to 'relu' function except the output layer that has 'softmax'. It is common to use softmax for the output since this function normalizes the output of a network to a probability distribution over predicted output classes.
The model is compiled with the loss function of binary cross entropy. [JRD20] has trained the model for 300 epochs. However, I used only 10 epochs due to the lack of computing tools, which I am resolving now (I am setting up my

Colab, and if necessary, other computing tools in the future). The results of these 10 epochs are shown in the table 3. Additionally, the diagram of the evolution of the loss value on the train and validation dataset for these 10 epochs is shown in the figure 20.

Table 3. Result of trained architecture after 10 epoch

Results	Results after 10 epochs
accuracy over train data	0.4849
loss over train data	1.3438
accuracy over validation data	0.4207
loss over validation data	1.5130
accuracy over test data	0.4232
loss over test data	1.4899

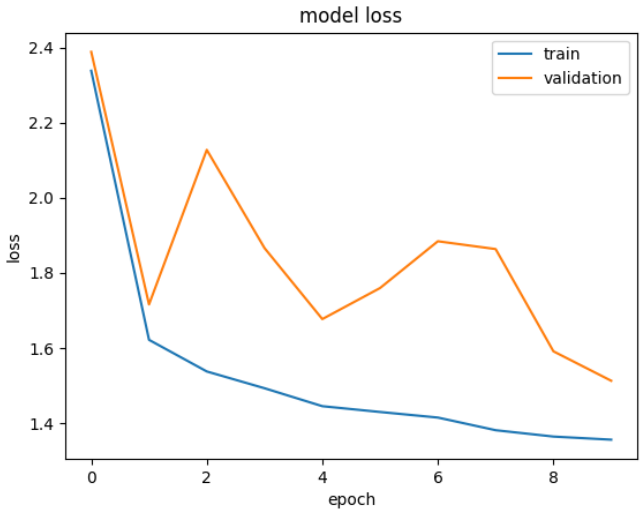


Figure 20

The code is available in github/fatemeafshar.

5. Face Expression Recognition using Neural Networks 2

This chapter is focused on the face expression recognition problem. Face expression recognition aims to capture the emotion of a face, which can be useful for numerous applications. There have been many algorithms devised and proposed to solve the face expression recognition problem. The most common one that has been very popular in recent years is neural networks. Neural networks are very suitable because of their ability to dig deep and extract features and especially the convolutional neural networks that are used in particular for image data.

This chapter implements an article [KC21] that has classified faces expressions using the neural networks algorithm. The dataset is taken from kaggle similar to the previous section.
The architecture of neural networks used in [KC21] is shown in the figure **??**. The architecture uses the VGGNET neural network, which is a classical model for large-scale image processing. The architecture comprises 4 stages. Each stage consists of two convolutional neural networks, a batch normalization layer for speeding the learning process and a max pooling layer to reduce the size. After these four stages, three fully connected layers are applied. The last layer classifies the 7 emotions. All the layers use a relu activation function, except the last one that uses softmax. It is common to use softmax for the output since this function normalizes the output of a network to a probability distribution over predicted output classes.

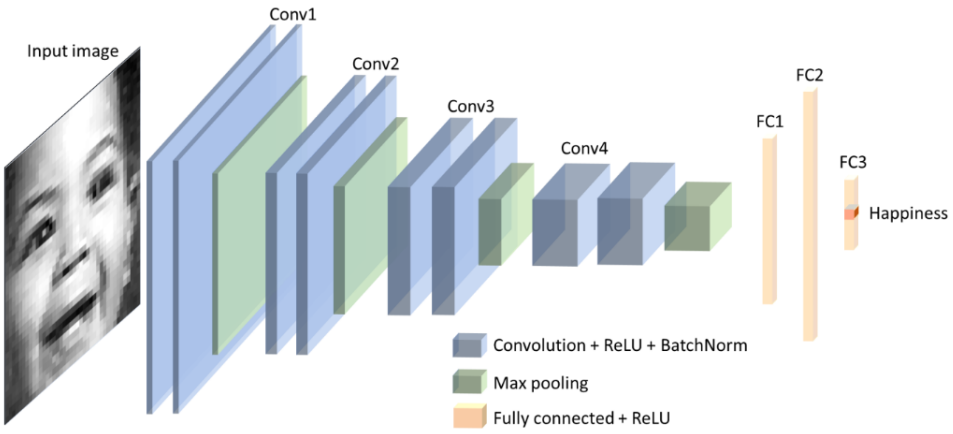


Figure 21

The [KC21] trains the model with various optimizers, including different variants of the SGD. The variant that is used in this section's project is SGD with parameters momentum= 0.9 and weight-decay= 0.0001. [KC21] also tests multiple approaches for scheduling the learning rate. The approach in this project is to initialize the learning rate to 0.01. The learning rate is reduced 0.75, if the validation accuracy plateaus for 5 epochs. To implement this, callbacks are used. There is a general method to schedule the learning rate that is shown in figure 22. In this code, the callback of learningRateScheduler is used that runs the function of scheduler in each epoch and updates the learning rate if necessary.

For the case of the validation accuracy checking, tensorflow has a special callback that is utilized in this project. The code for this callback is shown in the figure 23. This code sets the factor of decrease to 0.75, and it is going

```
def scheduler(epoch, lr):
    ...
    return learning_rate

scheduler_callback = tf.keras.callbacks.LearningRateScheduler
       (scheduler, verbose = 1)
```

Figure 22

to decrease the value of learning rate after 5 epochs of plateau of validation accuracy.

```
plateau_callback = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_accuracy', factor=0.75, patience=5, verbose=1
)
```

Figure 23

This callback is added to the parameters of training of the model as it is shown in the figure 24.

```
history = vgg16_model.fit(
    train_dataset,
    epochs =300
    validation_data = val_dataset,
    verbose = 1,
    callbacks=[plateau_callback]#scheduler_callback()]
    )
```

Figure 24

[KC21] has trained the model for 300 epochs. However, I used only 10 epochs due to the lack of computing tools, which I am resolving now (I am setting up my Colab and if necessary other computing tools in the future). The results of these 10 epochs are shown in the table4. Additionally, the diagram of the evolution of the loss value on the train and validation dataset for these 10 epochs is shown in the figure 25. As you can see in the table4, the

Table 4. Result of trained architecture after 10 epoch

Results	Results after 10 epochs
accuracy over train data	0.8539
loss over train data	0.1316
accuracy over validation data	0.4210
loss over validation data	0.4559
accuracy over test data	0.5702
loss over test data	0.3302

accuracy on the train data is good but the results on validation and test are not, which means that the model has not generalized. This network is a very heavy one with a large volume of neurons, which makes it very vulnerable to overfitting. The partial solution is to use some dropouts, but the better solution is to use larger dataset with methods like data augmentation.

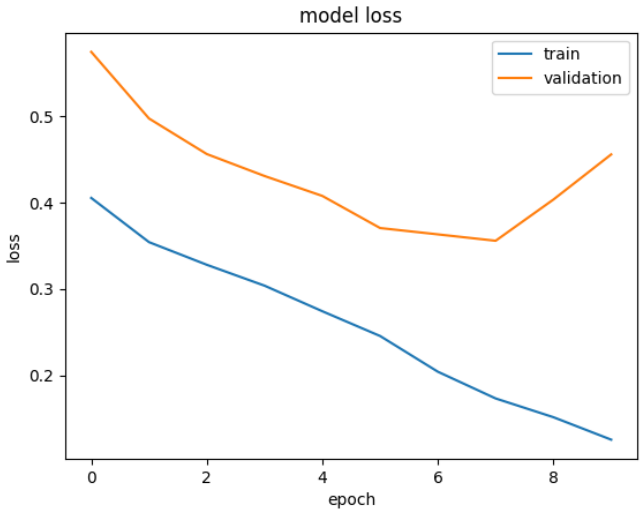


Figure 25

The code is available in github/fatemeafshar.

6. Face Expression Recognition using Delaunay Triangulation and Machine learning algorithms

In this section, another approach is used to solve the face expression problem. This approach is not based on neural networks, rather it uses other machine learning algorithms integrated with Delaunay triangulation. This is an implementation of the article [AA21] that uses Viola-Jones algorithm for face detection, active appearance model for shaape model and orthogonal procrustes analysis for normalization, to get the landamarks of the face(important points in a face). The facial landmark localization is defined as the localization of specific key points on the frontal face, such as eye contours, eyebrow contours, nose, mouth corners, lip, and chin. The overview of their work is shown in figure 26. According to the 26: First, landmarks are found, then Delaunay triangulation is applied on these points. The features are extracted from the triangles, and finally, a machine-learning model trained using the extracted features to classify faces.

The only difference between their work and this project is that they use statistical models and methods to find landmarks. However, this project, uses

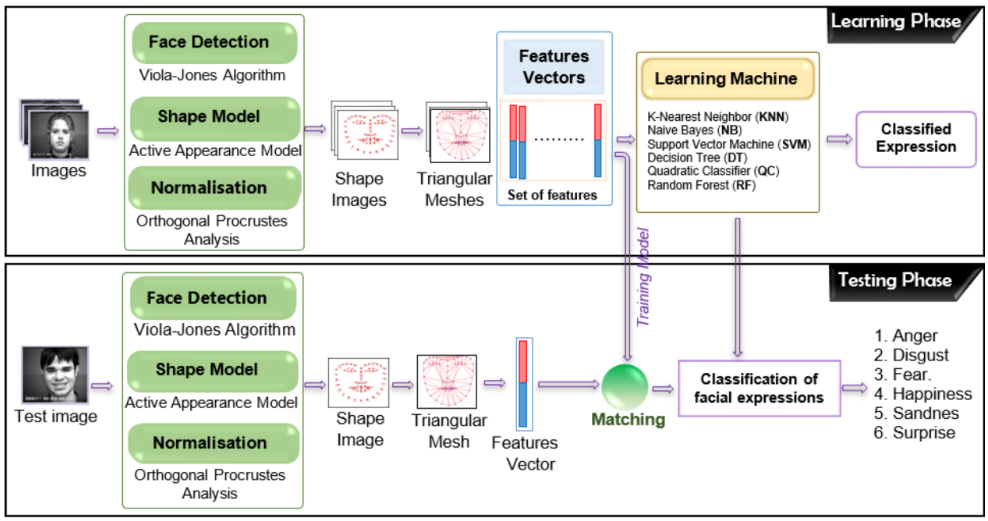
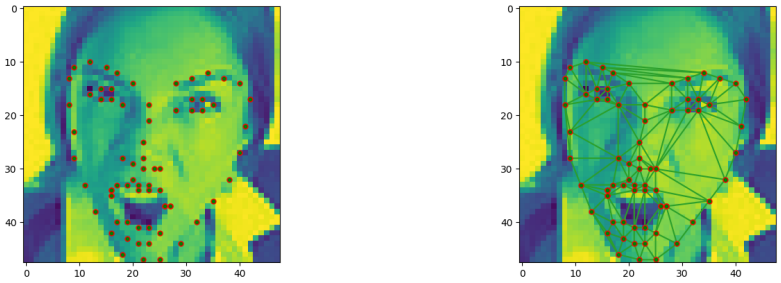


Figure 26. The overview of the [AA21].

the library of face_recognition that features a model to find the 71 landmarks. The result of 71 found landmarks is shown in the figure **??**. After triangula-



(a) 71 points of landmarks **(b)** Delaunay triangulation applied

Figure 27. finding landmark of a face and triangulation of it

tion, the article proposes two kinds of features that can be extracted from it. The first one that is also mainly used in the article itself is to consider the points and centeroid of each triangle as input features for the machine learning models. The other approach is to consider the angles of the triangles on the face. The article claims that the second approach did not prove to be very efficient in tests, so it utilized the first approach, but this project tests this approach too.

Afterward, six different learning algorithms are used to train triangulation data. These models are trained using the sklearn library. These machine learning algorithms consist of K-nearest neighbors, naive Bayes classifiers, support vector machine (SVM), random forest, quadratic classifier, and decision tree,

K-nearest neighbors (KNN) is a type of supervised learning algorithm used

for both regression and classification. KNN tries to predict the correct class for the test data by calculating the distance between the test data and all the training points. Then select the K number of points which is closest to the test data.
In statistics, naive Bayes classifiers are a family of linear "probabilistic classifiers" which assumes that the features are conditionally independent, given the target class. The strength (naivety) of this assumption is what gives the classifier its name. These classifiers are among the simplest Bayesian network models.
A support vector machine (SVM) is a supervised machine learning algorithm that classifies data by finding an optimal line or hyperplane that maximizes the distance between each class in an N-dimensional space.
A decision tree is a decision support hierarchical model that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements. Random forest operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees.
The accuracy is not yet very good and, for some algorithms, the type of features extracted makes a difference in different way. The inefficiency of results might be because the features have no locality and models can not generalize. I am working on the added features and interwokings of the models to get better results.

The code for this section is accessible via github/fatemeafshar.

7. Face Recognition using Delaunay Triangulation integrated with PCA

This section tries to solve another problem in the area of computer vision. Face recognition is one of the primary and foremost challenges in the area of computer vision. In face recognition, the model that has been trained by a large dataset of images, is capable of identifying faces. One of the most important and effective algorithms that is used for this problem is neural networks. However, before the rise of the neural networks, image classification was resolved by a method called PCA.
Principal components analysis (PCA) provides a data-driven, hierarchical coordinate system to represent high-dimensional correlated data. It reduces the size of features, which can be helpful for classification problems. PCA is implemented using eigen values decomposition(EVD) and singular values decomposition(SVD).
[AE20] implements PCA with SVD integrated with Delaunay triangulation to solve the face recognition problem. The Delaunay triangulation is a tri-

angulation that is always projected on a plane and has no two edges in the embedding cross [DB00], additionally, in this triangulation, a set of points in the plane subdivides their convex hull into triangles whose circumcircles do not contain any of the points. This maximizes the size of the smallest angle in any of the triangles.
To devise a face recognition classifier using PCA by implementing SVD; first, all the images in the dataset should be reshaped into columns in a matrix. The resulting matrix includes all the pictures in the dataset in its columns. Then, SVD is applied to this matrix and three matrices are achieved. To import svd, the library of scipy.linalg is used.

$$SVD = U \sum V^T \tag{1}$$

Where U and V^T are orthogonal matrices of eigenvector, and $\sum$ is the diagonal matrix of eigenvalues. Each column of the matrix U contains the information that can be used to generate an image, but the image is a corrupted one. Furthermore, SVD states, the information in the matrix can be approximated by a chosen number of the best eigenvalues. First, we should take the first k columns and transpose it. Multiplication of this matrix and an image(reshaped to a column) will result in an eigenface that holds the important information of a face.

$$eigenface = U_k^T.reshape(image, (w * h, 1)) \tag{2}$$

where k is number of eigenvalues that are used to approximate the image. [AE20] uses 25 eigenvalues to approximate the matrix of faces and generates eigenfaces. In the training phase, this multiplication is done for each image and the result is saved and has less size than the original dataset. If the dataset contains n images with the size(w,h). The image matrix, which is a dataset has (wh, n) size. The U matrix that is achieved from SVD has the size (wh, wh). Therefore, the matrix UT_k has size of (k, wh). Each eigenface has the size of (k,1), which means the new dataset has only the size of (k,n), which is much lower than the initial dataset's size (wh,n).
For classifying, the test image should be projected utilizing the first k=25 rows of U transpose (the eigenface of the test image is computed). Then, the Eucleadian distance between the test eigenface and each train eigenface is computed, and the least distance is chosen as the label of the face recognition process. This is because, if the eigenfaces of one person are drawn in a diagram according to their eigenfaces data, images owned by one person are clustered.
In the figures 28, 29 and 30 the k the best eigenvalues with values of 12, 25, 50 are used. The Delaunay triangulation process consists of finding landmarks of the face. In this project, 71 landmarks are found using the face_recognition

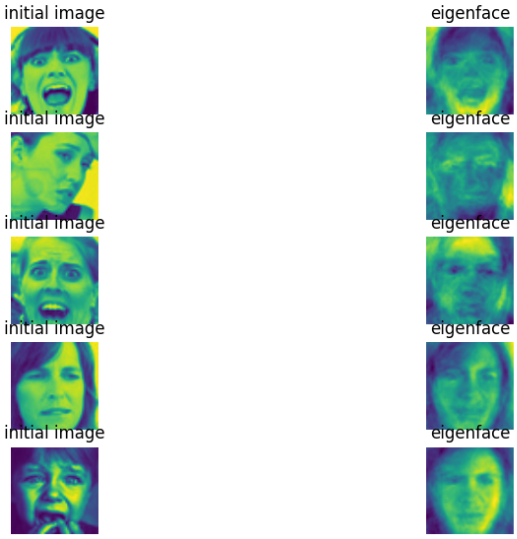


Figure 28. The first column: the raw image , the second column: the second column: matrix U multiplied to eigenface (to regenerate the image from eigenface which holds the most important data)

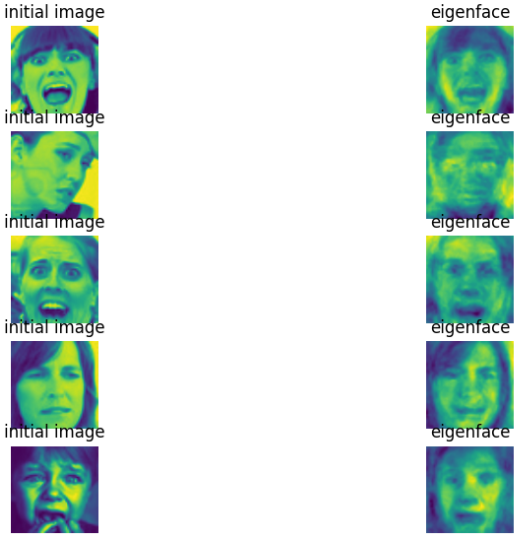


Figure 29. The first column: the raw image , the second column: the second column: matrix U multiplied to eigenface (to regenerate the image from eigenface which holds the most important data)

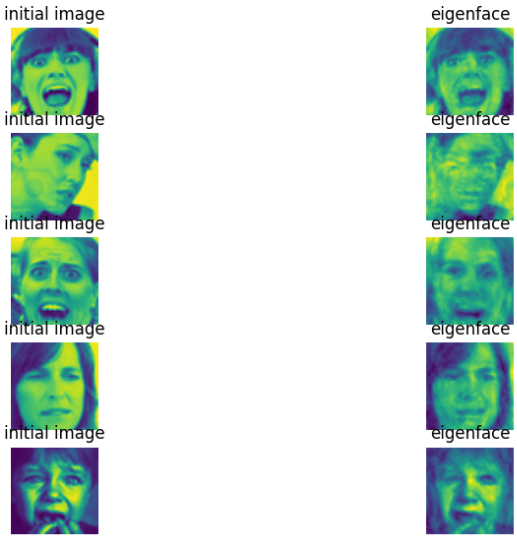


Figure 30. The first column: the raw image , the second column: the second column: matrix U multiplied to eigenface (to regenerate the image from eigenface which holds the most important data)

library. Moreover, to implement the Delaunay triangulation, the library of scipy.spatial is used. In the figure 31, the result of finding landmarks and Delaunay triangulation is shown. After triangulation, each triangle of the

(a) 71 points of landmarks **(b)** Delaunay triangulation applied

Figure 31. finding landmark of a face and triangulation of it

face is considered, and its area is computed. Then, the relative area of each triangle on the face to the biggest triangle is computed. Finally, the mean of all the relative areas is computed and stored in the database. In the test phase, to classify the test image, the same process is done, and the mean of the relative areas of the triangles in the test image is computed. Then, the difference of this value between all the train images is computed. The less

the distance, the more likely that the image is a match. To integrate these two methods, [AE20] uses this formula to decide the appropriate train image as the answer to the classification.

$$RV = ED + \frac{D}{0.001} \tag{3}$$

Where, ED is the computed Eucleadian distance computed using PCA, and D is the distance value achieved by the Delaunay triangulation method. The train image with the least RV (relative value) is chosen and is used to classify the test image.
The results of three different people were investigated for three k eigenvalues, 12, 25 and 50, and the corresponding accuracy for each was 0.36, 0.41 and 0.40. As you can see after 25, there is no improvement in accuracy, so, 25 was chosen for this project as well. I am working on methods to normalize faces based on their pose of the face to improve the classification.

The code for this section is accessible via github/fatemeafshar.

8. Qsmix: Q-learning-based task scheduling approach for mixed-critical applications on heterogeneous multi-cores

A Q-learning-based task scheduling approach for mixed-critical application on heterogeneous multi-cores (QSMix) to optimize their main design challenges is proposed. This approach employs reinforcement learning capabilities to optimize execution time, power consumption, reliability and temperature of the heterogeneous multi-cores during task scheduling process. In QSMix, a reward function is defined to consider all target design parameters simultaneously and is tuned based on applying punishment for unwanted conditions during the learning. The learning process of QSMix is led by utilizing the defined reward function during constructing the Q-table for various execution scenarios. Afterward, the best solution is selected from the constructed Q-table based on the system's policy to achieve a near-optimal solution that meets the existing trade-offs among objectives while considering its constraints properly. To evaluate our proposed QSMix, several experiments are performed to show its effectiveness in finding appropriate solutions and its gradual behavior during learning process. Moreover, the performance of QSMix in terms of optimizing the target design parameters is compared to various related research. The results confirm that QSMix has average improvement about 9% over related studies in joint optimization of execution time, power consumption, reliability and temperature.

The publication is available in Qsmix: Q-learning-based task scheduling approach for mixed-critical applications on heterogeneous multi-cores.

9. Acknowledgements

All the projects are open and constantly improved.

References

AA21. Farid Ayeche and Adel Alti. Facial expressions recognition based on delaunay triangulation of landmark and machine learning. *Traitement du Signal*, 38(6), 2021. (cit. on pp. 19 and 20.)

AE20. Kavan Adeshara and Vinayak Elangovan. Face recognition using pca integrated with delaunay triangulation. *arXiv preprint arXiv:2011.12786*, 2020. (cit. on pp. 21, 22, and 25.)

BK22. Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2022.

BTA+15. Peter Burkert, Felix Trier, Muhammad Zeshan Afzal, Andreas Dengel, and Marcus Liwicki. Dexpression: Deep convolutional neural network for expression recognition. *arXiv preprint arXiv:1509.05371*, 2015.

DB00. Mark De Berg. *Computational geometry: algorithms and applications*. Springer Science & Business Media, 2000. (cit. on p. 22.)

JRD20. Akriti Jaiswal, A Krishnama Raju, and Suman Deb. Facial emotion detection using deep learning. In *2020 international conference for emerging technology (INCET)*, pages 1–5. IEEE, 2020. (cit. on pp. 14 and 15.)

KC21. Yousif Khaireddin and Zhuofa Chen. Facial emotion recognition: State of the art performance on fer2013. *arXiv preprint arXiv:2105.03588*, 2021. (cit. on pp. 17 and 18.)