

Emotion recognition from images using deep learning

MSc Student Michail Tamvakeras, Associate Professor Ergina Kavallieratou

Dept. of Information and Communication Systems Engineering
University of the Aegean, Karlovasi 83200, Samos, Greece

Abstract

The ability to predict emotions based on static or dynamic images has improved the computer vision (CV) and robotics fields and remains a major research topic. Computer vision is a task of detecting or recognizing objects and persons on images or videos (sequence of images). Another important task is to predict the emotion of a person's face, which is called FER (facial emotion recognition). For example, in health care, a device or robot can observe the state of a person and call the ambulance if the person looks to be sick. An application could play specific types of songs i.e. Soul, Blues, Rock, etc. based on the mood of a person or a suspicious person or weapon could be detected by cameras. These are only some application fields, but there are existing many more. The component to extract and classify visual information's are convolutional neural networks (CNN's), which amongst other things comprise filters (kernels) and have a shallow or deep layer structure. The explanation of such networks and their application in facial emotion recognition tasks is the major topic of this document. To compare the performance of different networks, two deep neural networks (DNN's) will be used. The performance will be compared to each other and to other state-of-the-art solutions. Both networks will be trained using a method called transfer learning in which pre-trained networks will be adapted and fine-tuned to predict the seven emotion classes (angry, disgust, fear, happy, sad, surprise and neutral). The dataset that will be used to train and evaluate the networks is the FER2013 dataset from the Kaggle competition, which comprises 48x48 grayscale images and is specially created for FER tasks. To provide enough data while training the networks, the training set will be augmented with additional images during the execution.

Contents

1. Introduction	1
2. Related Work.....	3
2.1 Introduction to Artificial Neural Networks (ANN's).....	7
2.1.1 Activation Function	9
2.1.2 Forward Propagation	10
2.1.3 Backpropagation.....	11
2.2 Introduction to Convolutional Neural Networks	14
2.2.1 Convolutional Layer.....	16
2.2.2 Pooling Layer	18
2.2.3 Fully Connected Layer	19
2.2.4 SoftMax Function.....	19
2.2.5 Loss Function for CNN's	20
2.3 Existing FER solutions using CNN's.....	21
2.3.1 FER using Inception modules	21
2.3.2 FER using Residual connections	23
2.3.3 FER using ensembling.....	24
3. Proposed system	25
3.1 Dataset Creation	28
3.2 CNN's to solve FER tasks	32
3.2.1 The VGG16 Network	33
3.2.2 The Inception-v3 Network	35
3.3 Training	38
3.3.1 Adapted VGG16 Network.....	41
3.3.2 Adapted Inception-v3 Network	44
3.3.3 Hyperparameter Setup and the Training Process	47
3.3.4 Underfitting and Overfitting.....	49
3.4 Regularization	52
3.4.1 Learning Rate Decay	52
3.4.2 Dropout.....	52
3.4.2 Batch Normalization.....	53
3.5 Evaluation.....	55
4. Experimental Results.....	57

4.1 Experimental Results of the Fine-tuned VGG16 Network.....	60
4.1.1 First Experiments	60
4.1.2 Best achieved Training Result.....	63
4.1.3 Result of the Evaluation	66
4.2 Experimental Results of the Fine-tuned Inception-v3 Network.....	69
4.2.1 First Experiments	69
4.2.2 Best achieved Training Result.....	72
4.2.3 Result of the Evaluation	75
4.3 Comparison	78
5. Conclusion.....	83
References	84
Appendix	87
Python Code of the FER2013 Dataset Creation	87
Python Code of the Custom Visualization Tools	91
Python Code of the Main Program.....	94
Python Code of the Facial Emotion Recognition Manager (FerManager).....	95
Python Code of the Base Class of the Networks (CNNNetwork).....	100
Python Code of the Fine-Tuned VGG16 Network (FerVGG16)	104
Python Code of the Fine-Tuned Inception-v3 Network (FerInceptionV3)	109

1. Introduction

It has been known for a long time, that convolutional neural networks (CNN's) provide the ability to extract visual information's like edges, lines, curves, etc. from images and do classification based of this information. One of the first CNN networks was the Alexnet [1] and the VGG16 [2] network. The first comprises eight layers, including five convolutional and three fully connected layers. A max-pooling layer follows the first two and the last convolutional layers. After each convolutional and max-pooling layer, a ReLU activation function gets applied. All these terms and how CNN's work, will be described in detail in section 2. The second CNN which is one of the networks that will be used comprises sixteen layers, including thirteen convolutional and three fully-connected layers. A max-pooling layer follows either after two or three convolutional layers. Networks consisting of more than one hidden layer are interpreted as deep neural networks (DNN's). The second network is the Inception- v3 [3], which is one of the state-of-the-art networks consisting of multiple inception block types, convolutional and max-pooling layers. The consideration to compare an old and a new network is to verify if an old one is still capable of solving complex tasks such as FER with high accuracy. Another very important point is that older networks do not need much computational power as the newer ones and are able to be trained faster and without as many resources as the newer ones. Whether a very deep network is necessarily better as a deep one will be examined in this work. The dataset that will be used to train and evaluate the networks is the FER2013 [4] from the Kaggle competition, which is one of the most famous competitions among others dealing with FER problems. The dataset comprises 28.709 training, 3.589 validation (named as a public test set) and 3.589 test images (named as a private test set) divided into seven emotion classes (angry, disgust, fear, happy, sad, surprise, neutral). Each image is a grayscale image of size 48x48 representing a face-centered and occupying the same space. The dataset gets provided from the Kaggle website as a csv file. Unfortunately, the training examples are not as many as needed to train the networks. Because of that, only the training data will be augmented with additional, flipped, rotated, and so on images while the training process is executed. The process of creating the augmented dataset will be described in a later section. There are plenty of other FER datasets like the CK+ [5] available, which comprises 327 images divided into the seven emotion classes and 118 subfields into 10 groups. The FER2013 and the CK+ datasets are both freely available for research purposes. There are also commercial datasets available like the CMU MultiPIE [6], which consist of a huge number of images (750.000 images subdivided into 337 different subjects) but occupies only 4 categories (angry, fear and sadness are missing). Unfortunately, to use these datasets, they have to be purchased. To test the networks, 24 unseen RGB images of different sizes will be used, which are provided from the Dept. of Information and Communication Systems Engineering of the University of the Aegean.

The remainder of this document is structured as follows. Section 2 does a short introduction into the deep learning technology, the convolutional neural networks and introduces related works for FER problems. Section 3 presents the system that gets developed, including the dataset augmentation, the used network types, the training, and the evaluation process using the 24 unseen images and the test set of the FER2013 dataset. Section 4 discusses the experimental results of the training, validation, and testing process. The results will be compared to each other and to the proposed state-of-the-art solutions. The last section concludes the project results and introduces further work that can be done to achieve better results.

2. Related Work

In this section, related projects according to FER tasks will be examined. At first, the conventional FER process gets described without applying deep learning. Sections 2.1 and 2.2 introduces artificial neural networks and convolutional neural networks, which are the main components in deep learning. In the last part of this section existing FER solutions and techniques will be shown. On one hand for FER tasks, work was done using 2D images [7] that means ignoring the head position and orientation, and on the other hand using 3D images [7], where the dataset consists of sequential images in different time stamps, to provide the possibility of extracting spatial information. For simplicity, the first method will be used because extracting spatial information from a sequence of images is a very complex task.

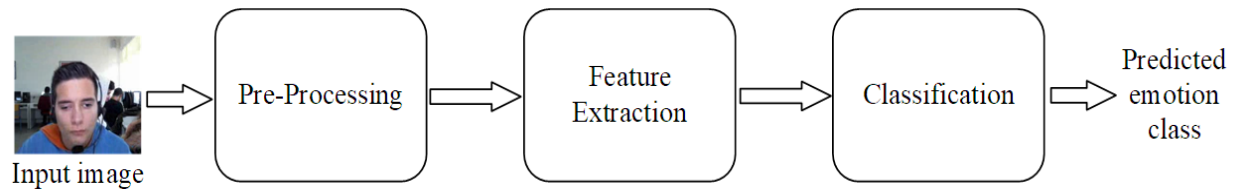


Figure 1: FER processes to classify an image into the emotion class

The FER process is divided into pre-processing, feature extraction and the classification step. In the pre-processing step, an image containing a face gets inserted, the face gets detected and the face region gets extracted using facial landmark points as (x,y) -coordinates. To determine if there is a face in an image, the Viola-Jones [8] algorithm gets used. This algorithm subtracts two regions of pixels from each other using so-called rectangular features. For example, when the eyelid is darker than the eye, so one or more of these features are able to detect this. To improve this process a machine learning algorithm using these features gets applied to a huge number of images to extract useless rectangular features, who are not able to detect these regions interest (ROI). The remaining features are used to determine if the image contains a face or not. If all these features return positive results, a face gets registered otherwise not. Figure 2 shows how these rectangular features look like and what they are able to detect. The figure, for example, shows that the second and third features are excluded by the machine learning algorithm, and all the remaining detect a facial part.



Figure 2: Rectangular features and what they are able to detect

Nevertheless, calculating a large group of images and summing them up takes too long and is a quite slow process. The solution is to calculate an integral image and use it for every image. Figure 3 shows how the calculation gets done. Every pixel gets added row by row and column by a column to create the resulting integral image. This gets then used to determine the value of an ROI, as Figure 4 shows. The region that gets subtracted twice, has to be added at the end of the calculation (in Figure 4 the purple box).

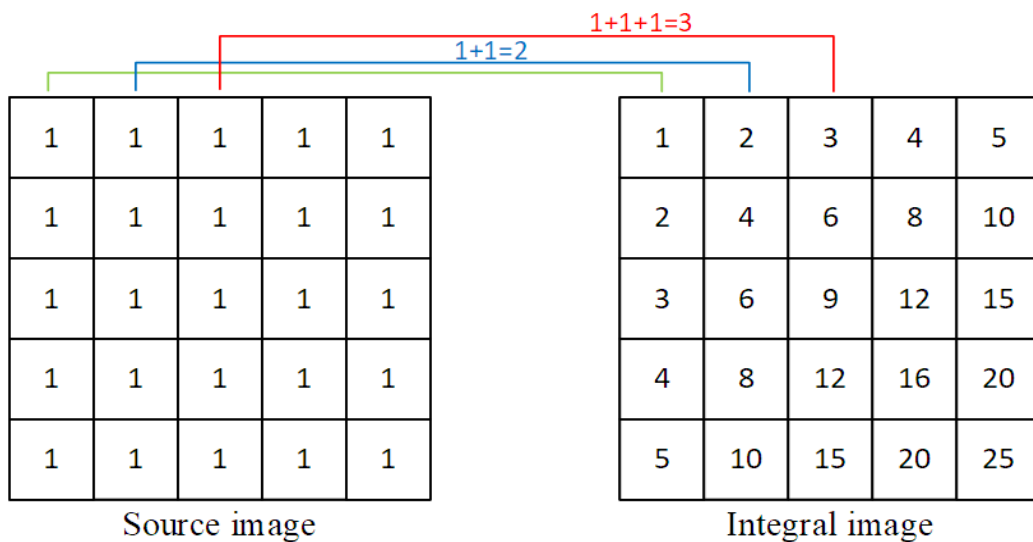


Figure 3: The creation of an Integral image

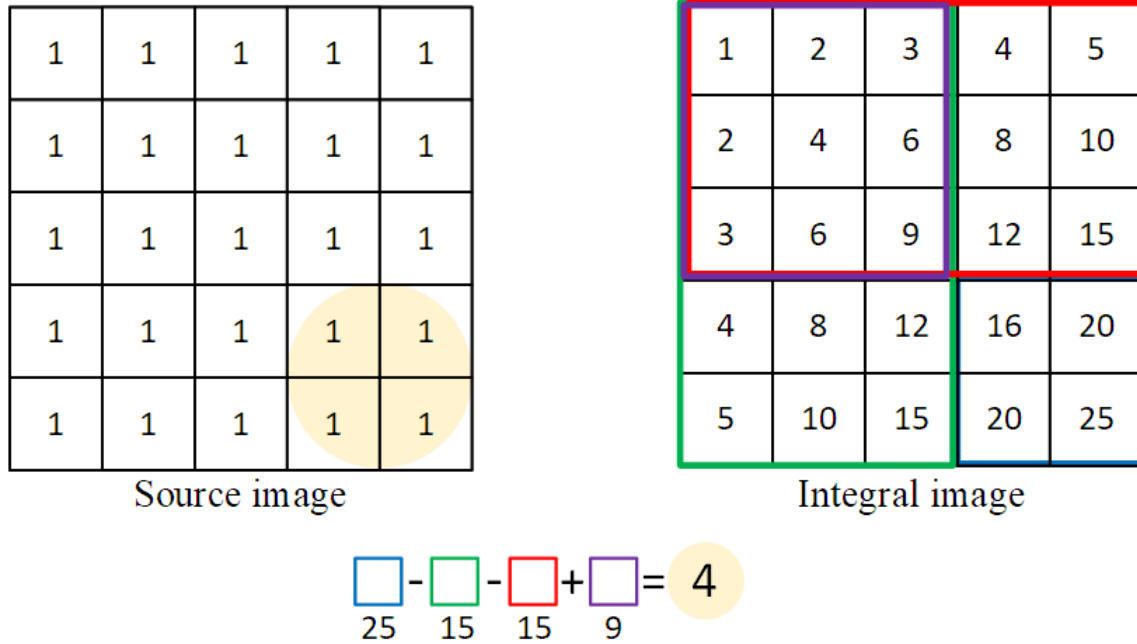


Figure 4: Viola-Jones region calculation

Based on the presented method, it is apparent that using an integral image, adding and subtracting regions occurs very quickly, without needing a lot of calculations of pixels. The Viola-Jones algorithm is extremely effective and gets still used, for example, in mobile phones containing a camera. The above method only detects if there is a face in an image. To extract the face region, landmarks are getting used to determine the locations of all facial parts (eyes, eyebrows nose, mouth, and the jawline). These landmarks provide additional the possibility to get the head pose if an eye is blinking and so on. The algorithm to extract these points is very complex, but fortunately, there is an implementation of the facial landmark detector that is freely available in the dlib Python library [9]. The detector is able to estimate 68 landmarks as (x,y)-coordinates which group facial parts together. Figure 5 shows how these landmarks are mapped to facial parts.



Figure 5: Viola Jones region calculation

Based on the predicted landmarks, the boundary box of the face can be determined and the ROI extracted. The face gets forwarded to the Feature Extraction step, where a different image processing algorithm is getting applied to extract useful features like corners, edges, lines, the brightness of pixels, etc. The last step handles the classification task which uses one or multiple machine learning algorithms like the k-NN [10], SVM [11] or Naïve Bayes classifier [12], to predict the emotion class based on the given features. The k-NN algorithm calculates the Euclidian distances between the data and groups them accordingly. The SVM classifier proceeds to higher dimensions to divide the data in a linear manner while the Naïve Bayes algorithm calculates probabilities to determine the classification. Unfortunately, all these methods do not provide any intelligence to learn anything during training time. In addition, all the data have to be pre-processed by hand, before they get fed into the classifiers which end in insufficient accuracy for the classification result of the emotions. To get around these problems, convolutional neural networks were introduced, which are able to learn the filters themselves and merge the feature extraction and classification step into one. The next section describes these networks and demonstrates at the end state-of-the-art solutions for applying these techniques in FER tasks.

2.1 Introduction to Artificial Neural Networks (ANN's)

Artificial neural networks are algorithms which act based on the human brain. They are able to classify nonlinear dividable data like the XOR function. Figure 6 shows how it looks like to divide linear and complex data. It is obvious that not all data points can be divided into the right class because this would overfit the data (overfitting will be described in section 3). One of the major advantages these networks provide is that after the training gets passed, new unseen data can be classified very quickly. The accuracy of these networks depends on many factors, such as the network type, the task it has to accomplish, and so on. Speed and accuracy are very important factors, especially for robotic vision tasks to reduce the delay of the interaction and to interact accordingly. There are different kinds of neural networks like the CNN's, which are dealing with images and videos, Recurrent Neural Networks (RNN's) [13] for speech recognition and Hybrid Neural Networks (HNN's) [14] for autonomous self-driving cars, self-flying planes, and drones. To solve the FER task, convolutional neural networks will be used, which will be described further in the document. Figure 7 depicts a deep neural network, consisting of its input, hidden and output layers. The input and the output neurons don't apply any activation function, as the graph shows. They only embody the input and the output of the network and don't execute any other operations. Each neural network can contain a huge number of layers and each layer can consist of multiple neurons. Networks consisting of multiple hidden layers are considered as very deep neural networks (VDNN's). In general, there is no number specified to consider a network as a VDNN. For example, the Inception-v3 network that will be used is a VDDN. The properties of these networks will be explained in section 3.

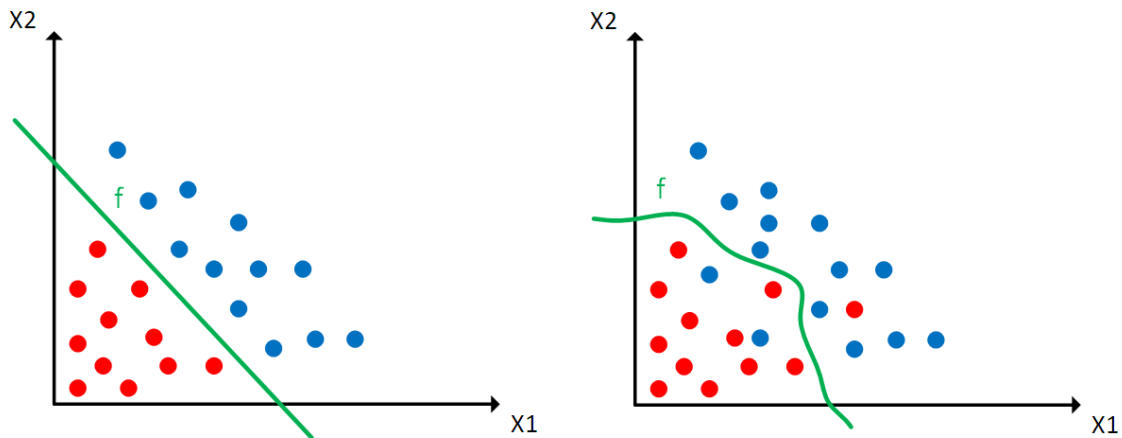


Figure 6: Left: linear dividable data, right: non-linear dividable data

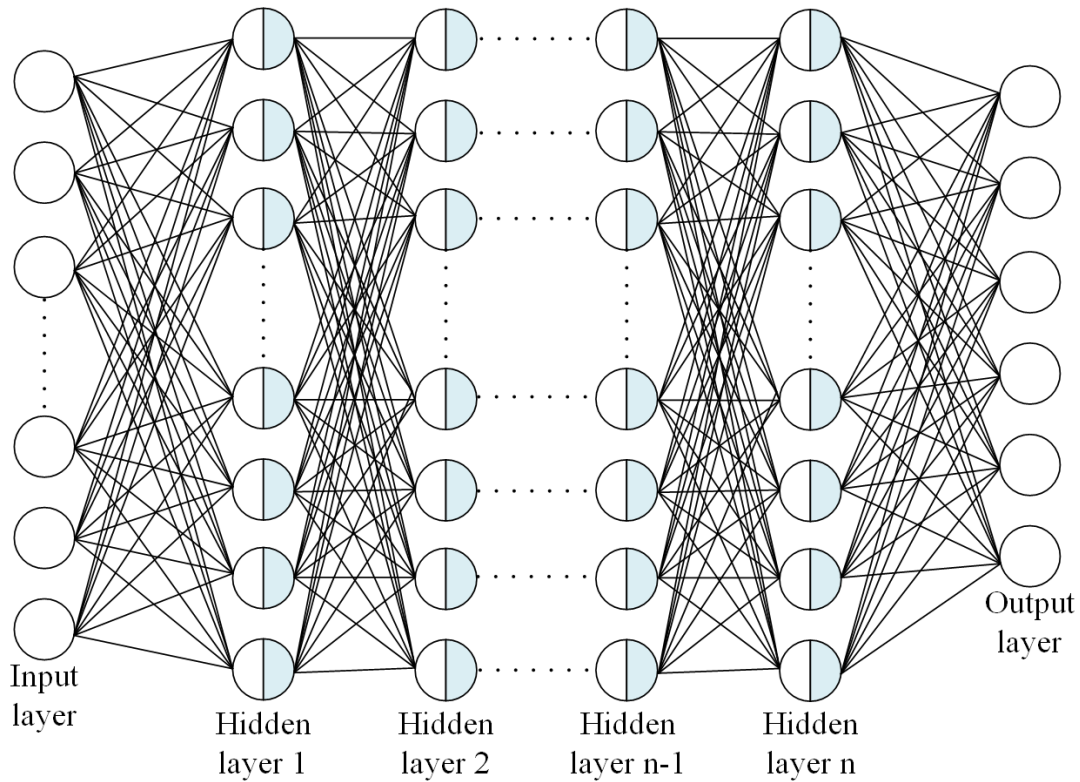


Figure 7: Deep neural network

A neuron calculates its output based on its input and its activation function, which is shown in Figure 8. In the net part, each input x_i gets multiplied by the weight w_i and the result gets summed up. In the end, a bias gets added to the summation. The input of a neuron can either be the input of the network or the activation of the previous layer. The out part calculates the output of the neuron based on the activation function and the result of the net part. The activation function shown in the Figure is the sigmoid function.

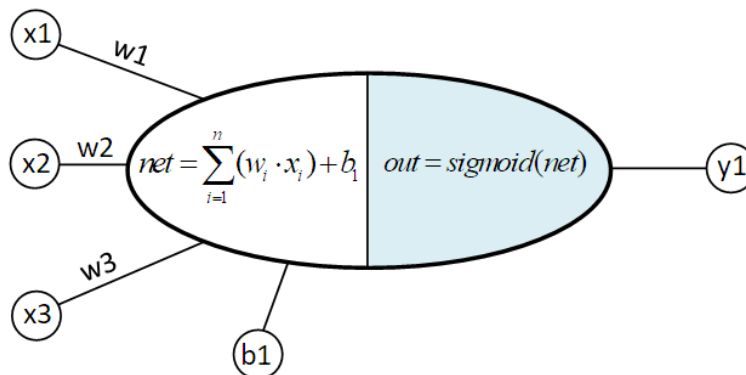


Figure 8: Neuron in an ANN

2.1.1 Activation Function

An activation function determines which calculated values of the net part will be passed through to the next layer. Figure 9 depicts some kinds of activation functions that are commonly used in practice. It is obvious that the sigmoid and the tanh function have the downside if the net values become either very large or very low, then the slope of the function gets close to zero, which slows down the gradient descent process extremely. The ReLU and the LeakyReLU activation functions are the most used functions in practice and in research papers. The first function cuts off all values lower than zero, whereas the second function allows passing some negative values. Nevertheless, the ReLU function gets more often used than the LeakyReLU, because cutting of the negative values decreases the computation amount (anything times zero results in zero) and increases the speed. This doesn't mean that the other activation functions don't get applied in applications. This depends from case to case and gets apparent through various experiments.

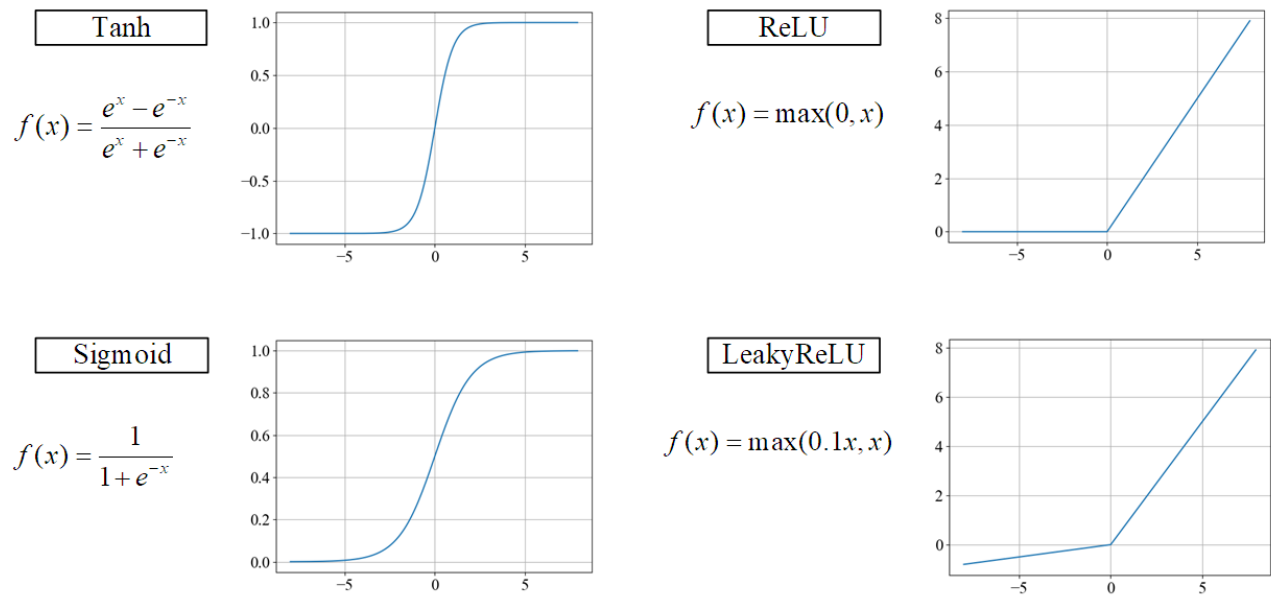


Figure 9: Activation functions

2.1.2 Forward Propagation

As mentioned previously, a deep neural network consists of multiple layers stacked in depth. This means that each neuron of the previous layer gets connected to each neuron of the next layer. Former layers are able to learn simple features, and as the network gets deeper and deeper, the network is able to learn more complex ones. Features are information provided based on the input of the network, that is, for image lines, curves, colors, etc. But how does a neural network work? In the first step the input data gets traversed through the network from the beginning to the end (forward propagation), each layer forwards its calculated output to the next layer and at the end, the network outputs its prediction. To evaluate the difference between the prediction and the ground-truth (target), a loss value gets calculated using a so-called loss function (sometimes called object function, too). There are a lot of different loss functions available dependent on the problem to solve. For example, the mean squared error loss (L2-loss) measures the Euclidian distance between the predicted and the target value and is defined as:

$$MSE = \frac{1}{2} (y_{pred} - y_{target})^2 \quad (1)$$

The variable y_{pred} contains the predicted value (label) and y_{target} contains the ground-truth value. The resulting difference gets squared to provide only a positive value. Section 3 introduces the loss function that gets frequently applied to calculate the loss of convolutional neural networks. The learning process executed in the opposite direction (backward pass) which will be explained next.

2.1.3 Backpropagation

The purpose of calculating a loss is to reduce the difference nearly to zero while training the network. Training in that sense means updating the weights and biases of the matrices, so the gradient descent algorithms are able to reach (if possible) a global minimum point (smallest possible point). This gets done by traversing the network in the opposite direction (backward propagation) and calculating the gradients from layer to layer using the chain rule. The computation to calculate the gradients is a very complex task and requires high mathematical knowledge in calculus. Fortunately, most deep learning libraries like TensorFlow and Keras do this work behind the scenes. The only thing the programmer has to do is to provide the loss function and all the backward calculations take place automatically (automatic differentiation). Figure 10 shows how the gradient steps traverse towards the global minimum (in most cases, there is not only a global minimum but one or more local minima. These are points that are minimal but greater than the global minimum). The steps towards the minimum get achieved using an algorithm called Gradient Descent (GD). There are many of these algorithms available, and each achieves the gradient step differently. Nevertheless, how big these steps are, gets determined by the learning rate α , which is, in other words, the velocity of each gradient step. If the learning rate gets chosen too large, the learning process may overshoot the global minimum, or if it gets too small, the training time will increase significantly. Figure 11 shows both of these cases, which concludes that the network will not be able to converge to a global minimum or waste too much training time. This makes clear that finding an accurate learning rate is essential to reduce the training time and to get high accuracy. It is not only important to find the best learning rate, but other hyperparameters as well, like the number of the epochs, the mini-batch size, the number of the hidden layers and neurons and the type of the activation function. These parameters get setup up at the beginning of each training process and are called hyperparameters because they determine the weights and biases of the network. How to regulate these and other hyperparameters to achieve accurate results will be described in section 3.

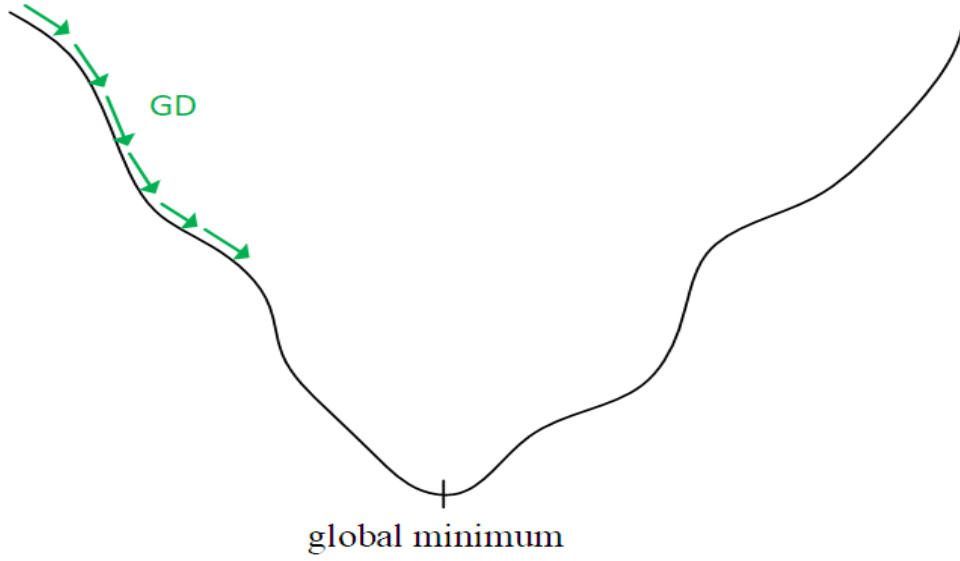


Figure 10: Gradient Descent Algorithm

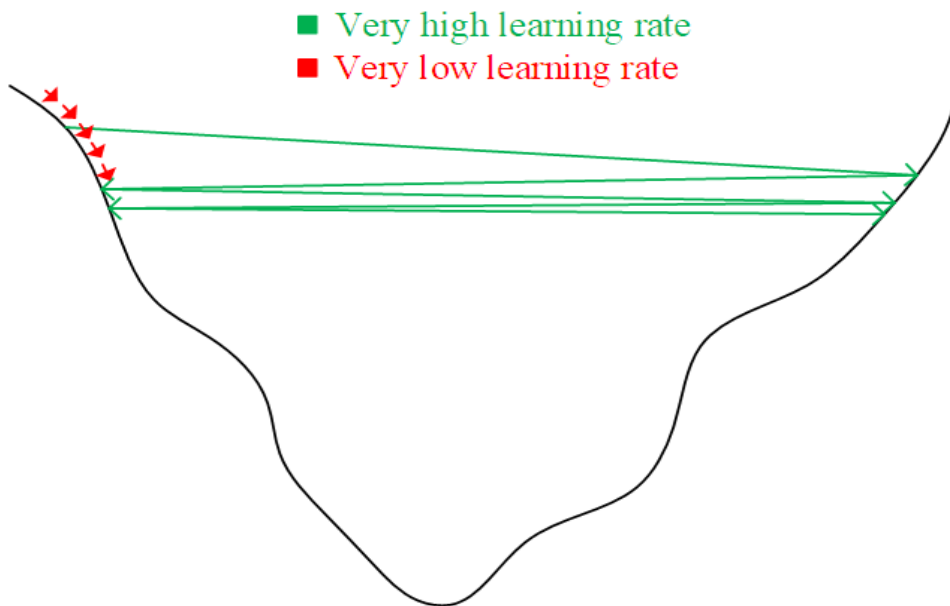


Figure 11: An example of bad learning rates

There are a lot of gradient descent algorithms available, and each of them has the responsibility to reduce the number of training epochs, providing high accuracy and make the network behave well. One of the first GD algorithms was the Stochastic Gradient Descent algorithm (SGD) [15], which selects the training examples in a random order and uses every iteration a batch size of one. This is not for every application suitable, because using only one example at a time does reduce the training speed significantly. Nevertheless, most deep learning researchers and practitioners use this algorithm because if work can be done well in a simpler manner, there is no need to look for additional solutions. In recent years, newer and more powerful algorithms have been developed like RMSprop [16] and Adam [16], [17]. Both of them are frequently used, whereas the second one is in most cases the first choice. The main goal of these algorithms is to speed up the learning process. The first algorithm reduces the number of the training steps towards the minimum (x-axis), whereas the second algorithm adds to the first one's operation a Momentum [16], which reduces additionally the oscillation on the y-axis. The behavior of the learning progress using a momentum combined with these algorithms is depicted in Figure 12. Without doing any optimization for the training steps towards the minimum, it is not possible to use higher learning rates to speed up the training process. The mathematical calculation of the gradients and how the updates of the weights and biases occur are described in detail in the respective paper. This work uses the Adam optimizer because it combines the possibility of the momentum together with RMSprop and achieves in most cases better performance.

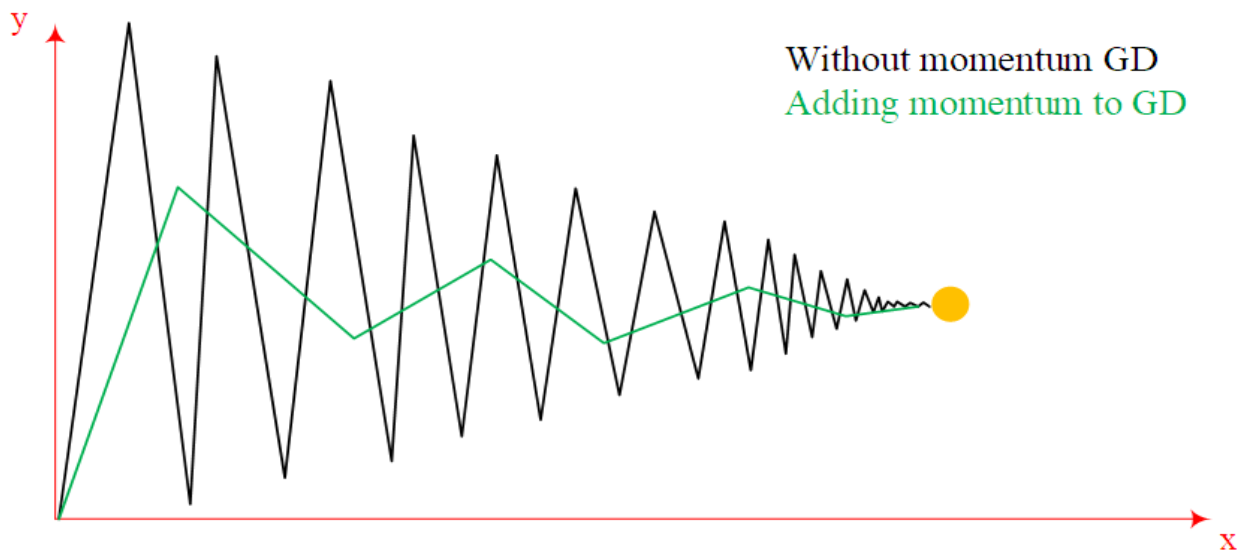


Figure 12: Learning axes for the visual representation

2.2 Introduction to Convolutional Neural Networks

As mentioned previously, there are many types of deep neural networks, and each is geared to accomplish special tasks. In order to solve the FER task, which is the main goal of this work, convolutional neural networks (CNN's) will be used. These networks can self-learn different kind of filters (i.e. edge and corner detector) and are able to extract visual features like edges, circles, or more complex like an eye, mouth, tree, etc. This excludes the feature extraction and classification step proposed at the beginning of section 2 and the training process gets “simplified” (the quotes denote that one hand the number of the process steps gets reduced, but on the other hand achieving high accuracy while using a CNN is not an easy task). In addition, the errors get reduced that can occur while hand-crafting the extracted features. Nevertheless, the pre-processing step remains, because on one side the images have to be resized appropriately to fit into the network and on the other side the faces have to be extracted. Using the previous method, the images can only be resized after the feature extraction step, which is not available when applying a CNN. There are cases where the CNN can be used as a feature extractor, too, which gets obtained, by cutting out the last layers after the last convolutional layer and forward these features for the classification. This gets not applied in this work, because the classification step would be added as a separate step, and this would only make sense if the extracted features would be needed as an input vector for a machine learning algorithm other than a CNN (i.e. an SVM or Naïve Bayes classifier, which are not described further, as they are not part of this work). The difference between the previous method and using a CNN to classify an input batch of images gets depicted in Figure 13.

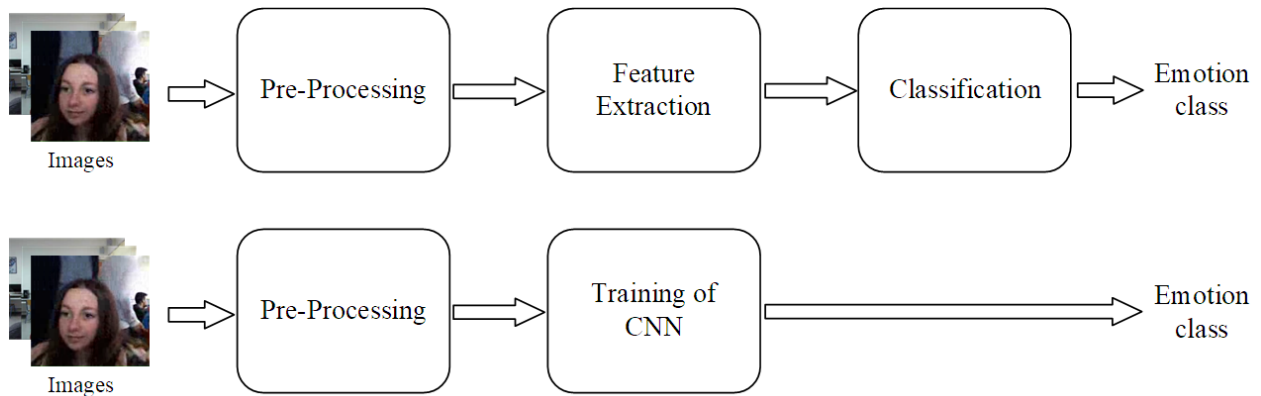


Figure 13: FER without deep learning (first row), FER using convolutional neural networks (second row)

Each CNN consists among others of multiple convolutional, pooling and fully connected layers, which are stacked in depth. The description of these layers will follow shortly. There are additional regularization techniques and components available which will be described in section 3. The first layers of a CNN are able to learn simple features like lines, curves, etc. whereas deeper layers use this combined information to learn more complex features like a tree, eye, face, etc. Figure 14 shows a simple structure of a CNN network, which consists of 2 convolutional layers, 2 pooling layers, one fully connected layer, and a Softmax function (the terms will be described in the next section).

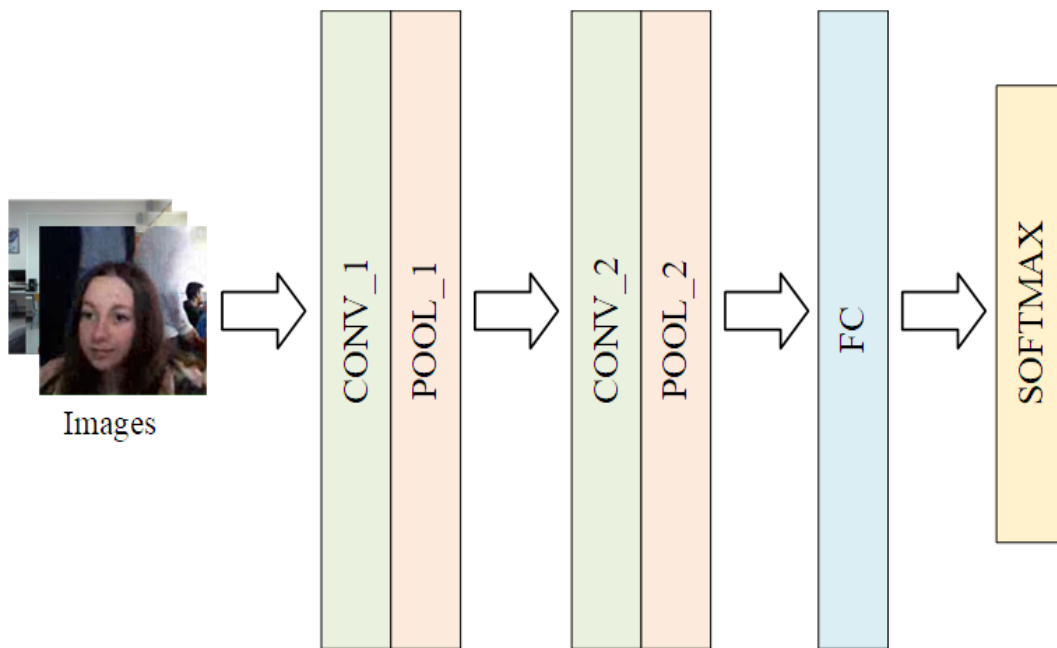


Figure 14: An example of a CNN network

2.2.1 Convolutional Layer

A convolutional layer consists of filters (kernel) who are getting applied to the input to calculate the result. Each filter starts from the top-left and gets shifted to the right pixel by pixel horizontally and row by row to compute the resulting matrix (feature map). Figure 15 shows an example of how these computations (also called convolutions) are executed. For simplicity, the computation gets done on a 5x5 grayscale image (M) and the values are mapped as follows: 1 for white, 0 for gray and -1 for black pixels. The filter (K) detects vertical edges and has a filter size (or kernel size) of 3x3. The mathematical formula to do this computation is the following where i is the row and j the column number:

$$C = \sum_{i,j} K(i, j) \cdot M(m+i, n+j) \quad (2)$$

The Figure shows that the filter is only able to travel three pixels to the right and three pixels down (without crossing the borders), that's why the resulting feature map (C) is of size 3x3. This concludes that the dimensions of the output image get squeezed, which is in some cases not desired. To keep these dimensions fixed, the borders can be expanded with zeros which is called padding (or more precisely zero-padding). The figure shows that indeed the filter detects a vertical edge (Figure 15 right: black-and-white image) on the image M. Executing the convolution operation on an RGB image follows the same process, with the difference that the calculation has to be done for each of the image channels i.e. red (R), green (G) and blue (B) and the filter has to be three dimensional, too. In general, the image channels have to match the filter channels e.g. if the image is 5x5x3 then the filter has to be 3x3x3, but the resulting feature map keeps of size 3x3 (1-dimensional). It is also possible to apply for each channel a different filter type to detect edges dependent to the color or to apply multiple filters at a time which results in a multidimensional feature map. For example, if ten filters are applied, then the resulting feature map ends to be 3x3x10. Doing all this computation while traversing deeper into the network, the resulting feature maps grow in depth, which means that for DNN's and especially for VDNN's the computation amount grows up extremely, which concludes that very powerful computers are required. One goal of research works is to reduce these computations to be able to construct VDNN's and to reduce the consumption of computational power.

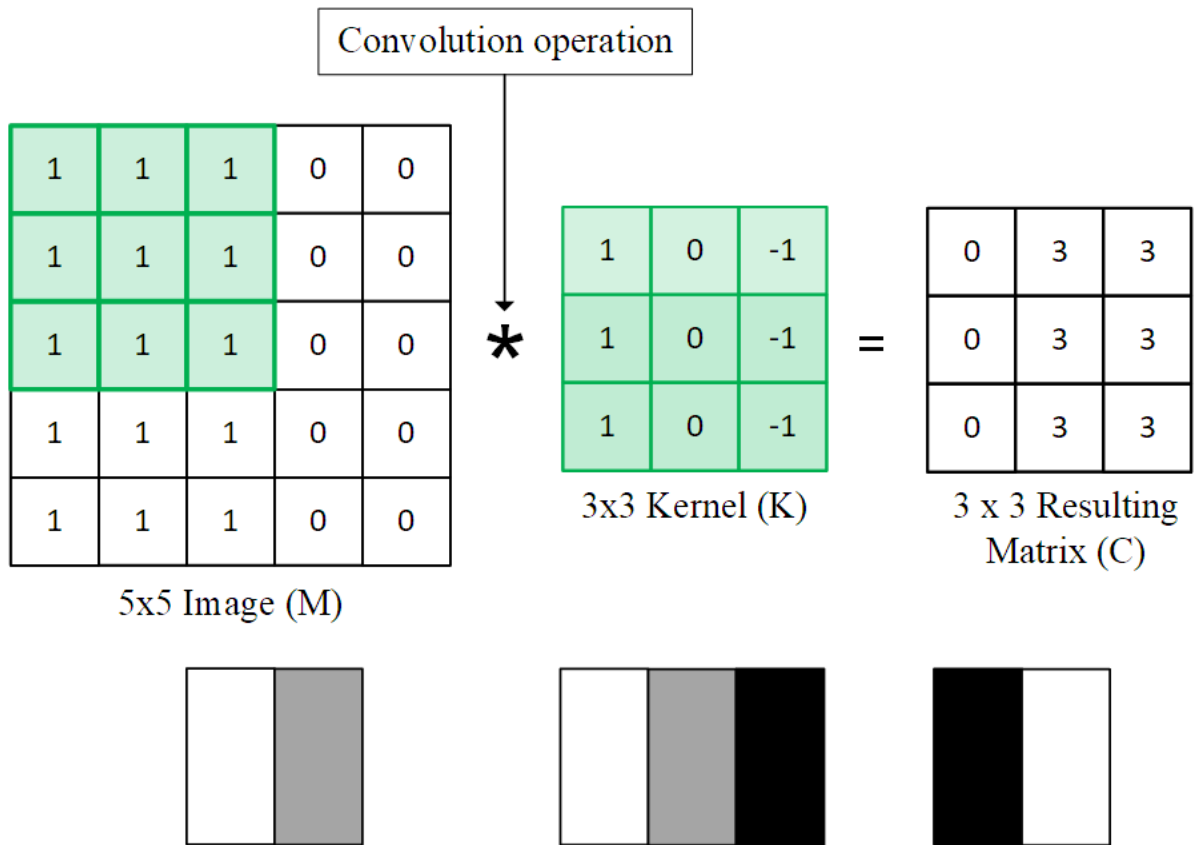


Figure 15: The convolution operation and the resulting image matrix

2.2.2 Pooling Layer

A pooling layer acts as a filter and reduces the number of computations while going deeper into the network. This gets achieved by reducing the dimensionality by calculating specific values. To do this, the stride of a filter denotes how many pixels the filter gets shifted to the right and down. The filter size and the stride are hyperparameters which get set up at the beginning and provides the advantage that the gradient descent algorithm has no parameters to learn. There are two pooling techniques available, one is called max-pooling and the other average-pooling. Max-pooling calculates the highest value of a region and average-pooling the average value. In some cases, the convolutional and the pooling layer are drawn as one (like Figure 14). Figure 16 shows the results of applying either max-pooling or average-pooling on a matrix (image).

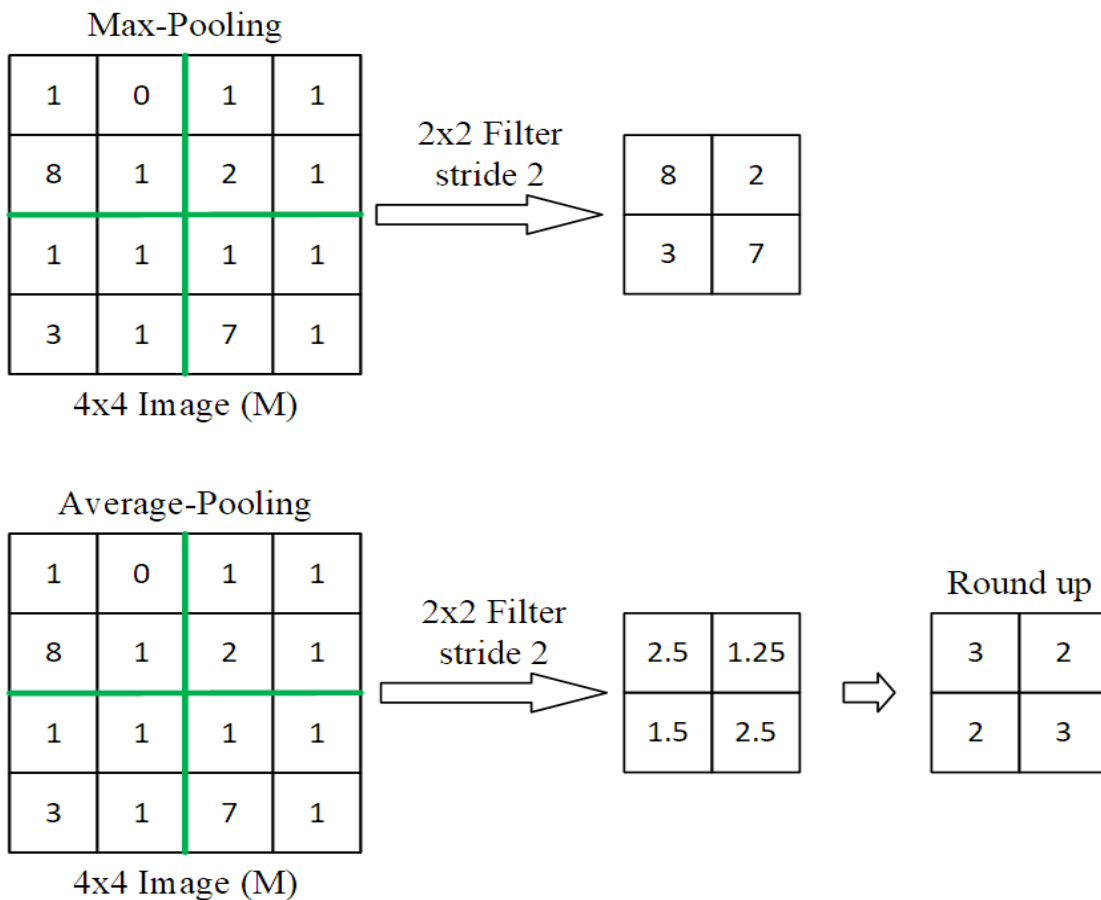


Figure 16: Pooling operations

2.2.3 Fully Connected Layer

A fully-connected layer (FC) connects each neuron of its layer to all the neurons of the next layer (hence the name fully connected). The layer is able to execute linear and non-linear operations and combines the features gathered from the last convolutional or pooling to create a one-dimensional vector. In case of there are multiple fully connected layers, the first one usually executes the ReLU operation, which is like doing a convolution on a 1-dimensional vector and the last one executes the SoftMax operation to predict the probabilities of the output. Using multiple fully connected layers reduces the required memory because the parameters forwarded from the last convolutional or pooling layer get reduced. Nevertheless, in contrast to a convolutional or pooling layer, it takes a very short time to compute the output because it doesn't need to compute a big number of weights like the others.

2.2.4 SoftMax Function

The last operation in the network architecture is the SoftMax operation, which takes the previous values of the fully connected layer and outputs for each class the probability. For example, if the last fully-connected layer consists of 1000 units that get feed into a SoftMax layer which outputs the probabilities of the seven emotion classes, then one output could be 0.2, 0.5, 0.1, 0.05, 0.05, 0.05, 0.05. This gives the second class the highest probability. Most competitions use these values to determine the top-1 and top-5 accuracy of a network or a network system. The top-1 accuracy determines how often the class with the highest predicted probability matches the target label divided by the number of evaluated examples, whereas the top-5 score determines how often the target label matches one of the five highest predicted labels divided by the number of the evaluated examples. The top-5 accuracy is not so strict like the top-1 accuracy, that's the reason why this value has a higher accuracy than the top-1 accuracy. A reason that there is the top-5 accuracy, is because there are similar classes which can not be distinguished e.g. the Angry and Disgust emotion are similar. Because this is the case in this work, both classes are getting merged into only one class while constructing the FER2013 dataset. The equation of the SoftMax function is defined as follows, where z_i is an element of the input vector z .

$$\text{soft max}(z)_i = \frac{e^{z_i}}{\sum_{i=1}^n e^{z_i}} \quad (3)$$

2.2.5 Loss Function for CNN's

To compute the loss of a CNN, the most used loss function is the cross-entropy-loss, which calculates the entropy of all classes and applies a SoftMax function to calculate the probability distribution of them. For example, in TensorFlow the loss function syntax `tf.losses.softmax_cross_entropy` denotes that the cross-entropy loss and the SoftMax function are getting applied together. The mathematical formula for the cross-entropy loss is the following, where y_{target} denotes the ground-truth label and p_i the predicted probability of each class.

$$Loss = -\sum_{i=1}^n y_{target} \cdot \log(p_i) = -\left(y_{target} \cdot \log(p_i) + (1 - y_{target}) \cdot \log(1 - p_i)\right) \quad (4)$$

2.3 Existing FER solutions using CNN's

This section introduces some state-of-the-art works that have been done in recent years, which use the above introduced CNN technology to classify facial emotions. There are approaches which use multiple filters combined in a block, another one uses special connections to bypass some operations and others use multiple CNN's to achieve a combined result. There are many more solutions available, and every time more and more are getting explored and developed.

2.3.1 FER using Inception modules

One of the introduced state-of-the-art technology of CNN's for FER tasks is the inception module [18]. This module consists of multiple filters of different sizes (1x1, 3x3, 5x5 and pooling), which are getting applied at once on its input. This makes, on one hand, the network more complicated, but on the other hand, this technique works very well because the network does not need to apply every iteration a different filter to find the right one which concludes that the epochs of the training get reduced. Because multiple filters are getting applied, and the result gets merged (concatenated) into a multidimensional feature map, all the output dimensions have to be of the same size. This gets obtained by applying padding to each output matrix. Figure 17 depicts the content of an inception module which consists of two 1x1, a 3x3, a 5x5, and a 3x3 max-pool filter. All these results get concatenated into one before the block forwards its result. As an example, if the first filter flow outputs 30 matrices, the second 15 and the max-pool 5, then the feature map has a depth of 50. Based on that, the next layer can self-choose or combine which of them it wants to use. Filters with smaller resolutions are able to detect local features on images like the nose, mouth, eyes, etc. and the bigger ones are able to detect global features i.e. spherical characteristics like the shape of the face or the chin. The inception module provides the advantage that the pooling performance arises, and so overfitting gets reduced significantly. Nevertheless, the computational cost grows, which in turn gets reduced by the 1x1 filter which reduces the dimensionality of the feature maps to the number of filters. In other words, this filter creates an intermediate volume (bottleneck) that has a reduced depth, which can be used for further computations to keep the dimension low. For example, if the input of the layer is of size 32x32x160 and 16 filters of size 8x8x160 with same padding (does the output image to have the same size as the input image, otherwise valid has to be used) get applied, then the resulting feature map is of size 32x32x16. This concludes an operation cost of $[32 \times 32 \times 16] \times [8 \times 8 \times 160] = 16384 \times 10240 = \underline{167.772.160}$, which is for a modern computer still very high. Using 16 filters, each of size 1x1x64 the cost gets reduced

to $[32 \times 32 \times 16] \times [1 \times 1 \times 160] = 16384 \times 160 = \underline{2.621.440}$. By a closer look, the depth gets reduced by a factor of 64.

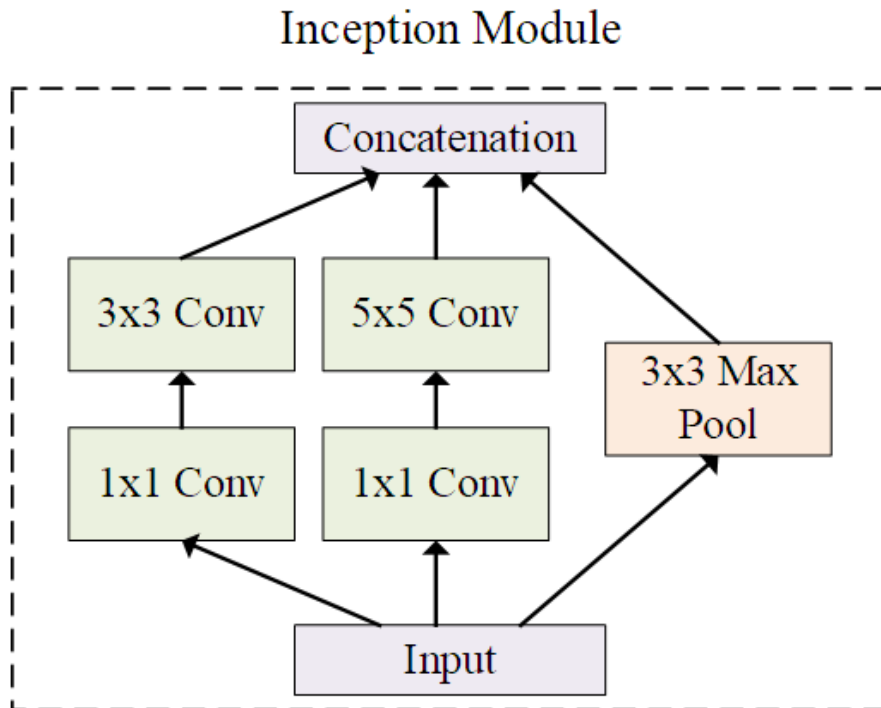


Figure 17: Inception layer content

The Inception network that comprises this technology achieved on the FER2013 dataset a top-1 accuracy of 66,4% and a top-2 accuracy (like top-5 but only for the two highest predicted probabilities) of 81,7%. One of the observations was that there were no emotions which couldn't be detected. The goal of the work was to show that including inception modules in a network, the detection accuracy improves as well.

2.3.2 FER using Residual connections

Another very neat solution to do FER with CNN's are residual-connections [19]. Very deep neural networks are in general very difficult to train, because of the vanishing and exploding gradients problem. This means in the first case, the gradient descent algorithm takes very tiny steps to learn anything, which leads to long training time. In the second case, the output value (y_{pred}) of the network gets very high, which increase extremely the loss. Nevertheless, no matter what is being developed for a new technology, the best results that can be achieved will be nearby the Bayes optimal error, which is the best possible error that can never be surpassed dependent on the problem. A residual connection (or shortcut) passes the result from a layer after the next layer as shown in Figure 18 shows a residual block. The example shows that the activation of the previous layer gets forwarded without getting modified before. This provides the possibility to execute the identity operation at that point. In the end, the operation gets added to the result of the convolutional operations, but before the next activation gets applied. This order matters because if the previous activation would be added after the next non-linear function, this value would not contribute to the overall result of the convolutional blocks. If multiple of these residual blocks get stacked together to a network, then this is called a ResNet [19]. In general, each CNN can be turned into a ResNet by inserting residual connections. A ResNet reduces the error while going deeper into the network. This is not the case for deep CNN's where these connections (or shortcuts) are missing. One type of ResNet is the ResNet-152 (152 is the number of layers) which was introduced by Microsoft Research and achieved a top-1 error rate of 21,43% and a top-5 error rate of 5,71%.

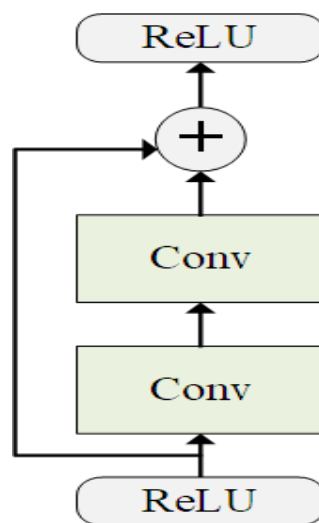


Figure 18: Residual connection

2.3.3 FER using ensembling

The process of combining two or more CNN's is called ensembling [20]. The system consists of two CNN networks, the deep temporal appearance network (DTAN) which gets trained on plain grayscale or RGB images and has the goal to recognize facial expressions with time. The temporal geometry network (DTGN), gets trained on facial landmark points and has the goal to recognize the movement of facial parts. Both networks get a feed with a small amount of input data for training. The prediction gets achieved by a third unit which they called the Joint-Fine-Tuning component. This combines both results of the networks to obtain the final prediction. The ensembling technique can be applied with two networks of the same type (like in the paper) and with different, too. The number of networks to use is not restricted, but using more parallel networks means more computational power for the computer system where the networks are trained. One solution is to use a cloud-based system or to distribute the computation on multiple computers. The goal of this work was to show that the ensembling technique can improve the accuracy of the emotion detection task. The details of which of the used datasets combined with the ensemble method achieved the best result can be obtained from the paper. For simplicity, Figure 18 shows a composite system consisting of two equal DNN's and an ensemble module. The input of the network is, on one hand, RGB images and on the other hand, other data like landmark points.

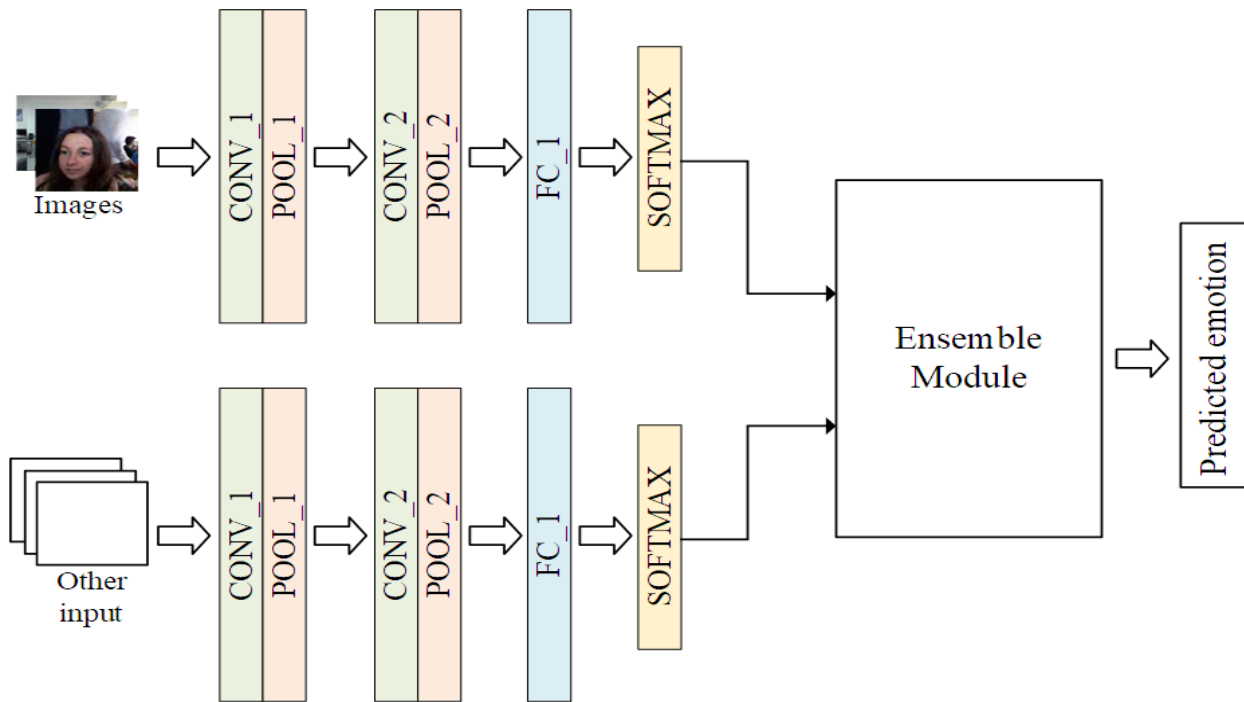


Figure 19: An ensembled system of two networks and the ensemble module

3. Proposed system

In this work, an emotion recognition system gets developed, consisting of one CNN, which takes as input grayscale images, that will be resized appropriately before they get fed into. The system gets developed on the operating systems Ubuntu 18.04.2 LTS and Windows 10, using the programming language Python and the development environment PyCharm Community Edition, which is freely available. The reason for using Python is the simplicity of the language and that it is an open source programming language with many open source libraries available for research and commercial purposes. The most important libraries are listed in Table 1.

Library	Purpose
NumPy	To calculate mathematical operations and conversions
Matplotlib	To plot the loss, accuracy, learning rate and confusion matrices
OpenCV	To apply image processing algorithms and display the images of the prediction
dlib	To extract the faces of the unseen images
scikit-learn	To calculate the confusion matrix
Pandas	To read the contents of the FER2013 dataset (CSV-file)
TensorFlow	To do low-level operations in callbacks
Keras	For all deep learning operations

Table 1: Python libraries used in this work

The training process gets executed on the windows system, which consists of an Nvidia GTX1080 GPU, 32GB RAM and an i7 CPU from Intel. To execute the training and prediction process, the deep learning framework Keras will be used, which is a freely available high-level deep learning API developed by François Chollet. Keras runs on top of either Theano [21], CNTK [22] or TensorFlow [23] (TF), which are low-level libraries and mostly implemented in Python, C, C++, and CUDA. The advantages of Keras are, that it is easy to learn, allows to prototype quickly and runs on both CPU and GPU very well. In case there is a need to execute low-level operations like tensor operations or GPU access, the Keras back-end can be used. Keras gets used as well for research as for commercial purposes. CUDA was developed by Nvidia, to allow parallel computing and to use resources on the GPU. Unfortunately, CUDA is only available for

Nvidia GPU's and doesn't support any other manufacturer like AMD. TensorFlow was developed by Google, CNTK was developed by Microsoft Research and Theano by the Montreal Institute for Learning Algorithms (MILA) at the University of Montreal. The project will be structured mostly in an object-oriented manner consisting of the files in Table 2.

Class	Description
FER2013DatasetCreation.py	Class to create the FER2013 dataset
MainProc.py	Main execution part, contains the networks creation and the management of the training and prediction process, which get handled by the FERManager
FERManager.py	Handles the GPU support, executes the training and prediction process
CNNetwork.py	Contains the common operations of the networks, such as reading the mini-batches, pre-processing the images and a custom callback class
FerVGG16.py	Contains the building, training, and evaluation operations of the adapted VGG16 network for the FER task
FerInceptionV3.py	Contains the building, training, and evaluation operations of the adapted Inception-v3 network for the FER task
VisualizationTools.py	Contains all the tools to set up and plot the losses, accuracies, learning rates and confusion matrices (optical more enhanced plot)

Table 2: The classes that will be developed

Two different CNN's will be used to compare the performance of a former and a state-of-the-art network in FER tasks. The reasoning, therefore, comes from that newer CNN's consist mostly of a very deep layer structure, which needs a lot of computational power and slows down the training process. Nevertheless, they are in most cases able to detect more complex features and achieve a better classification accuracy than former ones. The results of the experiments and the comparison against other state-of-the-art solutions will be discussed in section 4. The FER dataset that will be used, is the FER2013 dataset from Kaggle as mentioned earlier. Unfortunately, this dataset doesn't contain enough images to omit the overfitting problem while training the networks. Because of that, each mini-batch of the training process gets augmented with additional images. The process to create these images will be described in detail further in the document. To test the system, 24 unknown RGB images will be used, which are provided by the Dept. of Information

and Communications Systems Engineering of the University of the Aegean. These images do contain faces that are not aligned appropriately and therefore need to be extracted using the dlib library. The remainder of this section is structured as follows. In Section 3.1 the process to create the training set using the FER2013 dataset gets described. Section 3.2 presents both of the CNN types, that will be examined for the FER task. The whole training process using the adapted networks will be explained in section 3.3. Section 3.4 describes different regularization methods to reduce overfitting. Lastly, the evaluation process gets described.

3.1 Dataset Creation

There are plenty of datasets available which are suitable for facial emotion recognition purposes. Most of them are freely available for academic and private use. Table 3 lists the most used FER datasets consisting of the seven emotion classes Angry, Disgust, Fear, Happy, Neutral, Sad and Surprised. As mentioned earlier, the emotions Angry and Disgust are not easily distinguishable and that's why they get merged into one class. This concludes that the overall number of emotion classes will be six instead of seven. The advantage is a better learning process for the networks because the inaccurate predictions of the inaccuracy of these classes don't exist anymore. This method is also preferred in this work.

Dataset	Number of images	Type	Particularities	Provided from	Notes
CMU MultiPIE [6]	750.000 images	RGB	337 different people, 15 viewpoints under 19 illumination conditions	Carnegie Mellon University	4 emotion classes only available; the dataset is not freely available
MMI	2900 videos, more than 1500 images	RGB	25 subjects, static and image sequences, 75 subjects	mmifacedb.eu	6 emotion classes
CK+ [5]	593 images	RGB / Gray	123 subjects	PIA Consortium (CMU)	7 emotion classes, an extended version of the CK dataset
DISFA [24]	130.000 video frames	RGB	27 subjects, different ethnicities	University of Denver	7 emotion classes
FER2013 [4]	35.887 images	Gray	Already divided into training, validation and test sets	Kaggle competition	7 emotion classes, images for every emotion available

Table 3: Different FER datasets and their properties

The reasons to use the FER2013 dataset for the FER task are the following ones:

- The dataset is free for academic and private use
- The dataset consists of 48x48 grayscale images divided into the following datasets:
 - Training set: 29.709
 - Validation set: 3.589
 - Test set: 3.589

- Each emotion class contains more than thousands of images (except disgust)
- There are no empty emotion classes
- The faces are all aligned the same and they are all the same sizes

The FER2013 dataset is provided as a csv-file and consists of overall 35.887 grayscale images of size 48x48 divided into training, validation and test set. Unfortunately, training a deep neural network a lot of additional images are needed than provided by the dataset. Because of this disadvantage, each mini-batch before it gets feed into the network, these images get created to reduce the overfitting problem. In other words, the mini-batch gets augmented with additional flipped, rotated, zoomed, shifted, skewed, and so on images during training time. For that purpose, there is a convenient class available in Keras, which is called ImageDataGenerator and does all these operations automatically. The only thing the programmer has to do is to provide the ranges and modes for all of these operations. The images can then be used with the `fit_generator` (instead of the `fit`) method to train the networks. While creating the FER2013 dataset, each image and label gets stored in an hdf5 file. This file type acts like a database that facilitates very quick access and is specially designed to store and organize a huge amount of data. At first, the csv content gets imported into a data frame consisting of the column's emotion, pixels, and usage using the Pandas library. The library is open-source and is a convenient data analysis tool for Python, which provides operations to handle numerical tables and time series with high performance. While reading the contents of the csv file, they are divided appropriately into training, validation and test data. Because the image data gets read as a one-dimensional vector, they will be reshaped to the size of 48x48. It is not necessary to apply face extraction algorithms as mentioned in section 2, because the faces of the FER2013 dataset are already aligned appropriately. The above operations will be done using the OpenCV library for Python. The library is open-source and contains a huge amount of image processing and computer vision algorithms. Each image gets converted into a tensor (number, height, width, dimension) before it gets stored because a tensor can be handled better in linear algebra than a plain matrix. The emotion labels angry and disgust get merged into one emotion class because of similarity. Figure 20 depicts the overall creation process, and Figure 21 shows a sample of how the FER2013 images look like. The Figure shows additionally an example of image augmentation for the top-left image.

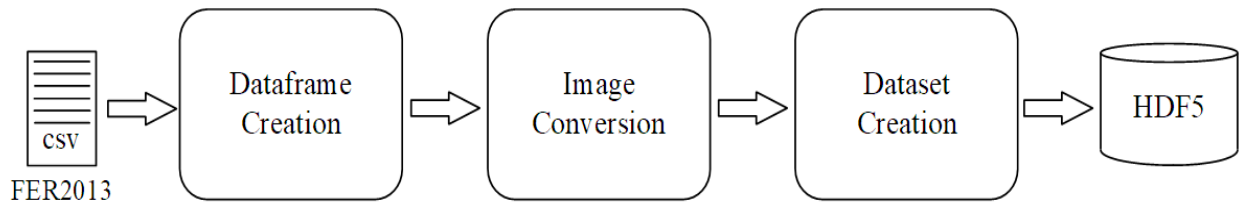


Figure 20: Creation process of the FER2013 dataset



FER2013 Images



Figure 21: Example of FER2013 images and image augmentation

The summary of the creation process is depicted in Figure 22, which shows, that the emotions Angry and Disgust got merged into one emotion class and the other emotions are shifted to the left (e.g. emotion Happy from 3 to 2).

The FER2013 dataset consists of:

```
-----  
[Training]      Images: 28709   (28709, 48, 48)   Labels: 28709   (28709, 1)  
[Validation]    Images: 3589    (3589, 48, 48)   Labels: 3589    (3589, 1)  
[Test]         Images: 3589    (3589, 48, 48)   Labels: 3589    (3589, 1)  
[Emotions]     0:Angry, 1:Disgust, 2:Fear, 3:Happy, 4:Sad, 5:Surprise, 6:Neutral
```

FER2013 dataset creation in progress...

Creating the training set...

Creating the validation set...

Creating the testing set...

FER2013 dataset creation completed.

Duration: 0:02:47

[NEW LABELS] 0:Angry/Disgust, 1:Fear, 2:Happy, 3:Sad, 4:Surprise, 5:Neutral

Figure 22: Creation results

3.2 CNN's to solve FER tasks

As mentioned in the introduction section, to execute the FER task, two pre-trained CNN's will be used in this work and the performances get compared to each other in section 4. The CNN's are pre-trained on ImageNet [25] which consists of RGB images of different sizes. ImageNet is the world's biggest competition for computer vision tasks and provides more than 1 million images divided into 1000 different categories for the participants, to train and test their networks. In contrast to the images of ImageNet, the FER2013 dataset consists of 48x48 grayscale images, which leads that the data doesn't fit into the pre-trained networks without pre-processing them before. To get around of this problem each image of the mini-batch gets resized based on the network input size, normalized and converted to an RGB image before it gets feed into the network. Fortunately, the OpenCV library provides operations, which accomplish this in only one line of code. If the input of the networks gets desired to be grayscale images, then the first convolutional layers must be retrained again. In general, it is recommended to use the same image types based on the pre-trained networks input. In this work, the CNN's will be trained on grayscale images which will be converted previously to RGB as described above. In addition, all images will be resized to 100x100x3 pixels, to reduce the number of computations and memory. The next section describes the networks VGG16 and Inception- v3 in detail and in section 3.3 the networks will be adapted and fine-tuned to solve the FER task.

3.2.1 The VGG16 Network

One of the first CNN's introduced in the ImageNet competition was the Alexnet [1] and the VGG16 [2] networks. Today's most computer vision tasks owe the introduction of these networks. The reasons to use the VGG16 network instead of the Alexnet, is on one hand that there is no pre-trained network available in the Keras library which was trained on a huge number of images, and on the other hand the VGG16 network uses mostly small filter sizes of 3x3, max-pooling layers of 2x2 with a stride of one or two and with a padding of same (the output size is the same as the input). This configuration reduces the amount of the overall parameters to train. The VGG16 network consists of 16 hidden layers (hence the name), but there is also the VGG19 [2] version available, which consists of 19 hidden layers, which will not be considered further in this work. The pre-trained VGG16 network on ImageNet consists of more than 138 million parameters as Figure 23 shows, which is still for modern PC's very large. Shortly different methods will be described to reduce this number so that the CNN can be trained on the proposed system. The advantage of the VGG16 network is its simple architecture of uniform layers as Figure 24 shows. There are two to three convolutional layers which are following by a max-pooling a layer. The max-pooling layer reduces the height and width of the input size. As can be seen, the number of filters gets doubled from convolutional block to convolutional block. Padding gets set for each block to same, to preserve the width and height as explained earlier. The term convolutional block refers to a unity of convolutions with a pooling operation. The inputs of the pre-trained network are RGB images of size 224x224x3. The last convolutional block is followed by three fully connected layers containing 4069 and 1000 neurons and at the end, a SoftMax function (will be described later) gets applied to predict the probability of each of the 1000 classes. The network achieved at the ILSVRC-2014 ImageNet competition a top-1 error rate of 24,7% and a top-5 error rate of 7,3%. Both of these measures denote the opposite of the top-1 and top-5 score. For example, the top-5 error (accuracy) determines how often the ground truth label doesn't match one of the five highest predicted classes divided by the number of evaluated examples.

```
=====  
Total params: 138,357,544  
Trainable params: 138,357,544  
Non-trainable params: 0  
--
```

Figure 23: Parameters of the pre-trained VGG16 model

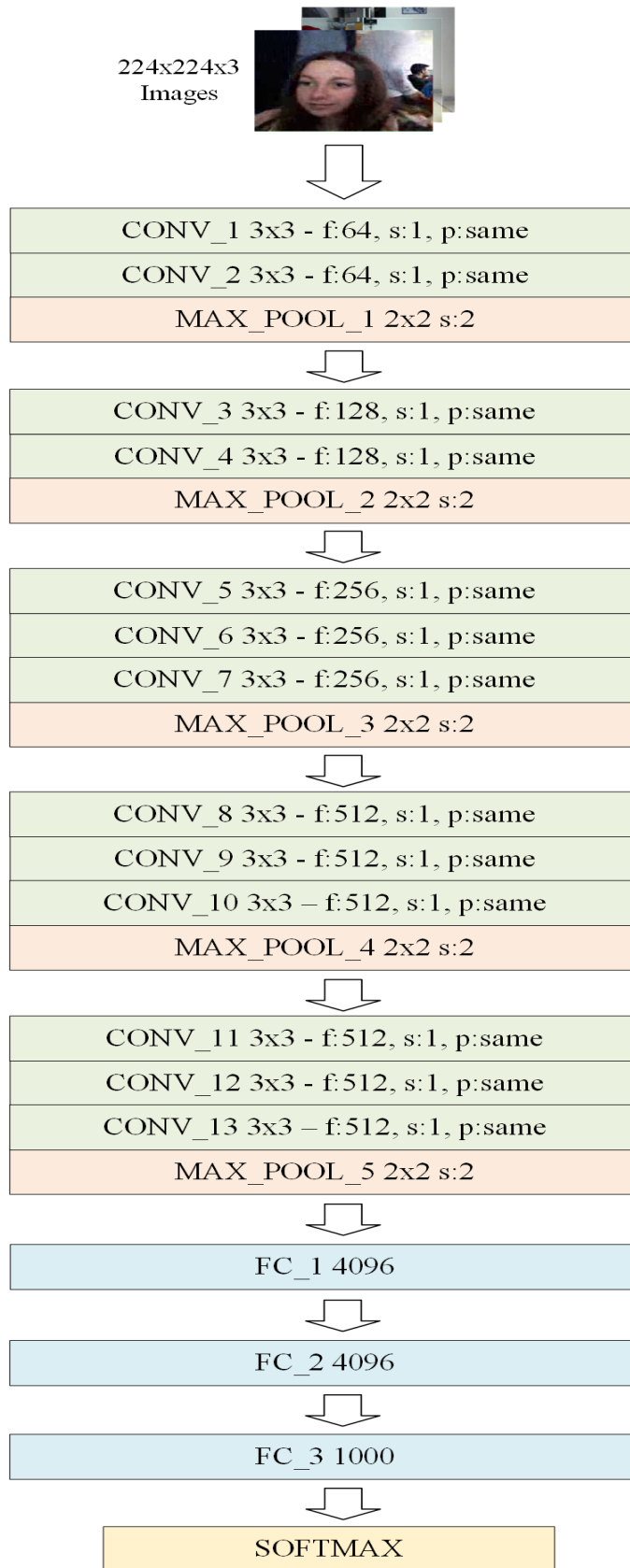


Figure 24: The architecture of the VGG16 network

3.2.2 The Inception-v3 Network

The Inception-v3 [3] network is the CNN that will be compared against the VGG16 network in this work. To decide which state-of-the-art network to use, a lot of other networks were tested like the Inception-ResNet-v2 [26] and the ResNet [19]. Unfortunately, it could be observed that using a very deep network was difficult to train on the provided system. The reason is, the big number of parameters to train and the huge number of the required training examples to omit overfitting. In contrast, the Inception-v3 network consists only fewer layers which in turn allows training the network with the provided dataset (including augmentation). The differences of them are that the Inception-ResNet and ResNet consist of residual connections and reduction blocks but the Inception-v3 network not. That is the reason for the huge number of parameters. The Inception-v3 network gets used for computer vision tasks and research purposes because it combines multiple filters at once in a block. The network consists of inception modules which got explained in section 2.3.1. The overall architecture is small and consists of six 3x3 convolutional and two pooling layers which are of size 3x3 and 8x8. The main components of this network wherever the name is coming from are the three different kinds of Inception blocks. All blocks contain four filter paths as can be seen in Figure 26. Each path contains different filters and each filter gets executed one after the other. At the end of each block, the results of all filter paths get concatenated and forwarded to the next component. The first inception block consists of 1x1, 3x3 and one max-pooling layer. The second block consists of 1x1, 7x7, 7x1 (column-vector), 1x7 (row-vector) and one max-pooling layer and the last one consists of 1x1, 3x3, 1x3, 3x1 and one pooling layer. Each of the inception blocks recognizes different kinds of features. The network was developed in collaboration with Google and the University College of London. The pre-trained network from ImageNet contains more than 23 million parameters as Figure 25 shows. Figure 26 shows how the three inception blocks internally look like. The overall network architecture consists of multiple inception blocks as Figure 27 shows. The result of the last inception block gets forwarded to the last max-pooling layer to reduce the output size. In the end, a SoftMax function gets applied to predict the probabilities of the 1000 classes used for ImageNet. The network achieved at the ILSVRC 2012 ImageNet competition a top-1 error rate of 4,2,9% and a top-5 error rate of 18,77%.

```
=====  
Total params: 23,851,784  
Trainable params: 23,817,352  
Non-trainable params: 34,432
```

Figure 25: Parameters of the pre-trained Inception-v3 model

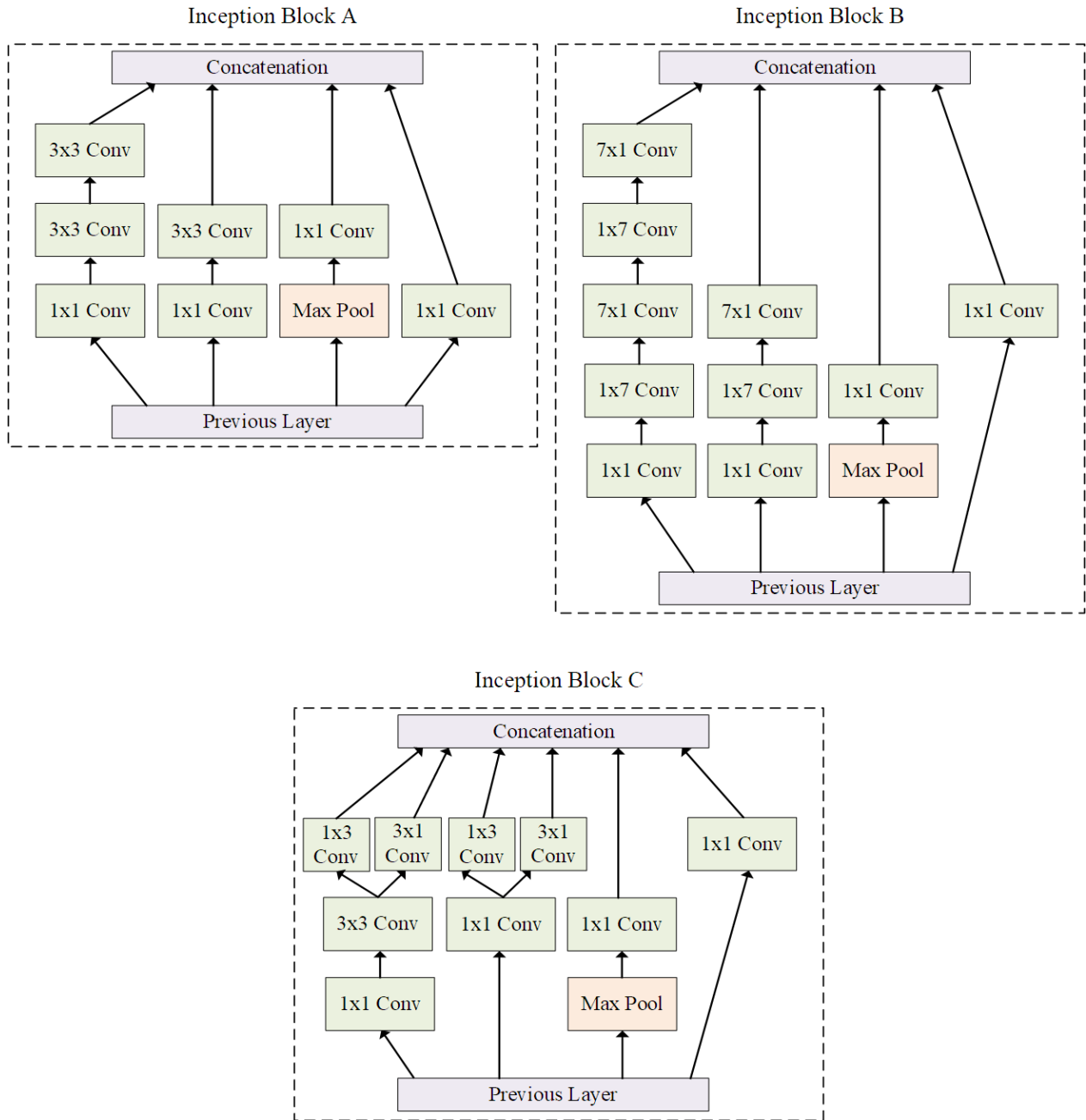


Figure 26: The three Inception blocks

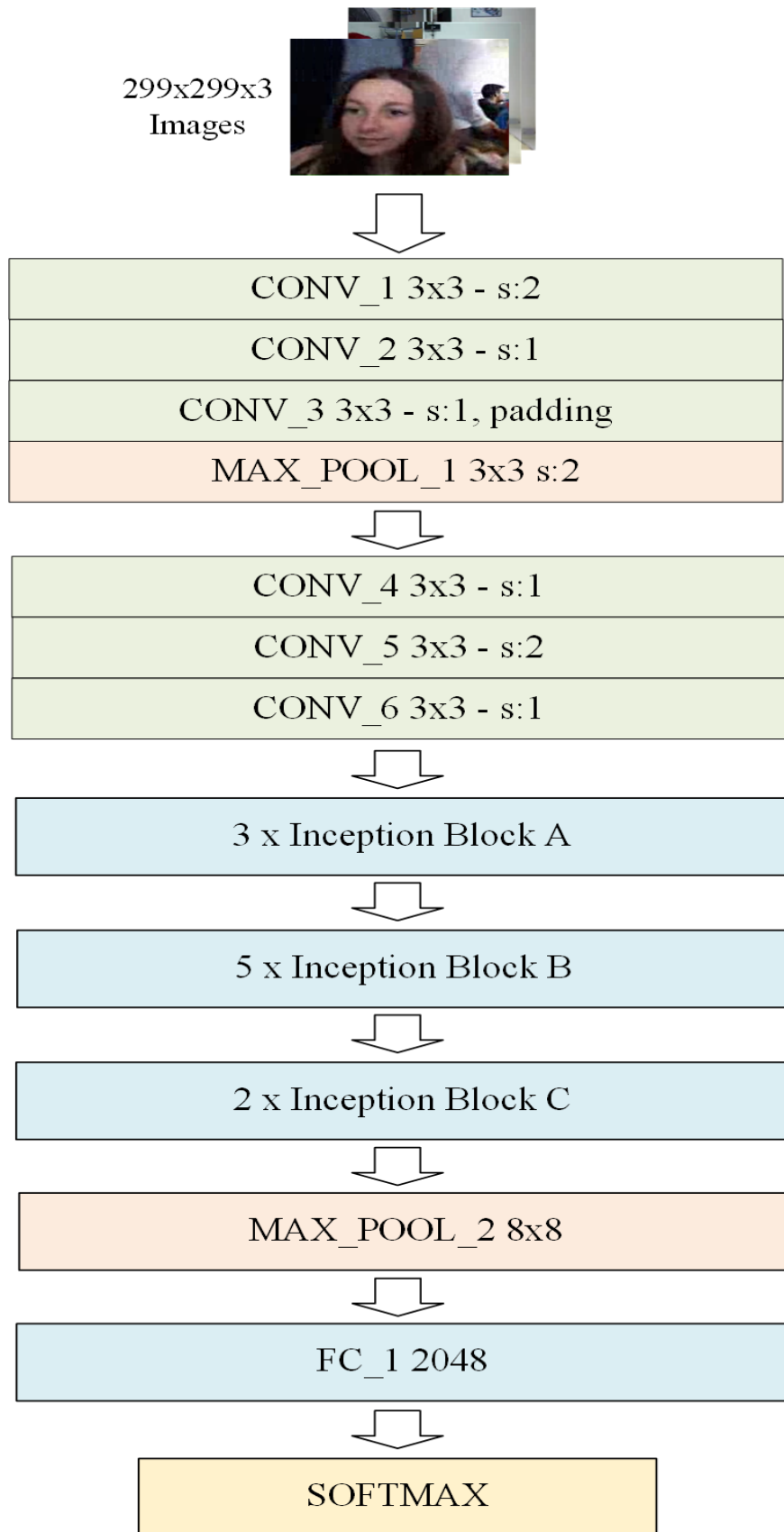


Figure 27: The architecture of the Inception-v3 network

3.3 Training

This section describes the whole training process of both networks using a technique called transfer learning, whereas the next section describes the testing process using the unseen images provided by the Dept. of Information and Communications Systems Engineering from the University of the Aegean. To train a deep neural network from scratch a huge number of images are required (500.000, 1.000.000 or even more images) to reduce the overfitting problem. The drawback of this operation is that the training time increases, the initial weights have to be initialized appropriately to get a better start of the gradients and a very powerfull computer system is required. To overcome those problems, the system gets trained using a technique called transfer learning by fine-tuning. This means that a network gets used, which was trained on a similar task, but containing another prediction layer (last fully connected layer). For example, all the networks of the ImageNet competition were trained on 1000 classes, which are of everyday life objects like cats, dogs, trees, devices, etc. In contrast in the project, the prediction layer has to be exchanged and retrained again to classify only the six emotion classes. Exchanging the prediction layer and re-training it is in most cases not enough to get a suitable performance for the respective task. In that case, it is necessary to unfreeze and retrain more layers of the network starting from the first one and continuing unfreezing additional layers until the best performance can be reached. The layer number to unfreeze depends on the amount of data and the similarity of the task to the task of the pre-trained network. Table 3 shows all the available transfer learning methods. For example, if the dataset is big enough and similar for the task to solve, then the pre-trained network needs only to be fine-tuned. On the other hand, if the data to use is big enough but different than the data used for the pre-trained models, the network has to be trained from scratch. As an example, could be a pre-trained model which was trained on images containing trees. Since all models in the Keras library were trained on ImageNet, which doesn't contain any images showing faces like e.g. in the FER2013 dataset, the pre-trained networks have to be fine-tuned from scratch and if necessary one or multiple layers have to be unfrozen. The cell with the blue color shows the method that will be used in this work.

Transfer Learning methods		Data similarity	
		None	Similar
Size of the data set	Big	Training the whole network from scratch	Fine-tuning the whole network
	Small	Fine-tuning the lower layers only	Fine-tuning the last layers

Table 4: Transfer learning methods

Using transfer learning makes it possible to fine-tune the weights and biases appropriately (in other words the backpropagation process gets executed from the last point the network was saved, and not from scratch) and do not need to be retrained from scratch, which saves a lot of time. In most cases, it is enough to freeze all layers starting from the beginning until the last pooling or convolutional layer and let all the remaining ones to be trainable. Another case is to freeze only the first layers and let all the remaining be trainable. This is because the first layers of the pre-trained models were already trained to recognize simple features like edges, curves, etc. Nevertheless, if the goal is to achieve high performance and the dataset is big enough the best to do is to keep all layers trainable. In general, transfer learning by fine-tuning doesn't require as much training images like training from scratch which in turn reduces significantly the training time. Applying transfer learning is a frequently used method in research and professional projects. It still needs to be mentioned that training a network either from scratch or by applying transfer learning is an experimental process which requires a lot of time, experiments, patient and knowledge of the deep learning technology. Each network used in this work gets adapted and fine-tuned appropriately. The adaption includes among others, adding layers, or adding regularization techniques (will be described later) like dropout and batch normalization. Before each mini-batch gets feed into the network, the images have to be resized to a size of 100x100x3 as mentioned earlier to use bigger mini-batch sizes. This, in turn, allows the gradient descent algorithm to calculate better gradients instead of mini-batches of small sizes. Based on the Keras documentation these dimensions are valid because for the VGG16 network the minimum allowed size is 32x32x3 and for the Inception-v3 75x75x3. To intervene into the training process different callbacks can be used. Callbacks are functions which execute a specific operation on a specific task.

For example, the callback History saves all the events into a history object. The callbacks that will be used in the project are ModelCheckpoint to save the network weights of the best validation accuracy, EarlyStopping to stop the training if a minimal loss of 0.001 gets reached and the LearningRateScheduler which decays the learning rate in specific time-stamps (will be explained in section 3.4.1). The Keras framework provides the possibility to create additional callbacks which get used in the project, too. More precisely a custom callback class called PredictionHistory will be created which saves all the predicted and target values of each mini-batch execution, to use them for the subsequent confusion matrix.

3.3.1 Adapted VGG16 Network

The best-adopted version of the VGG16 network to solve the FER task is presented in Figure 28. In general, it takes a lot of experiments and patience to reach a satisfactory adaption. The Figure shows that the layers from the first one until the penultima convolutional layer keep frozen and the following including the additional added ones are set to be trainable. To improve the performance of all layers after the last max-pooling layer the network gets adapted as follows. Batch normalization gets added to normalize the output of the max-pooling layer which keeps all the values in a specific range. To reduce overfitting the following dropout layer gets added to drop every iteration randomly 20% of the neurons. Each of the remaining neurons gets connected to the fully connected layer, which now only consists of the six FER classes. At last the SoftMax function gets applied to predict the probability distribution of the classes. It can be shown that the input size of the images is as mentioned 100x100x3 and not 224x224x3. Figure 29 shows the parameters of the pre-trained VGG16 network and the adapted version. It is obvious that the pre-trained network on 1000 classes contains more than 138 million parameters to train, which is by far too much. Instead, the adapted version (top right) shows that the parameters to train are only 14 million, which is round about 10 times lesser then before.

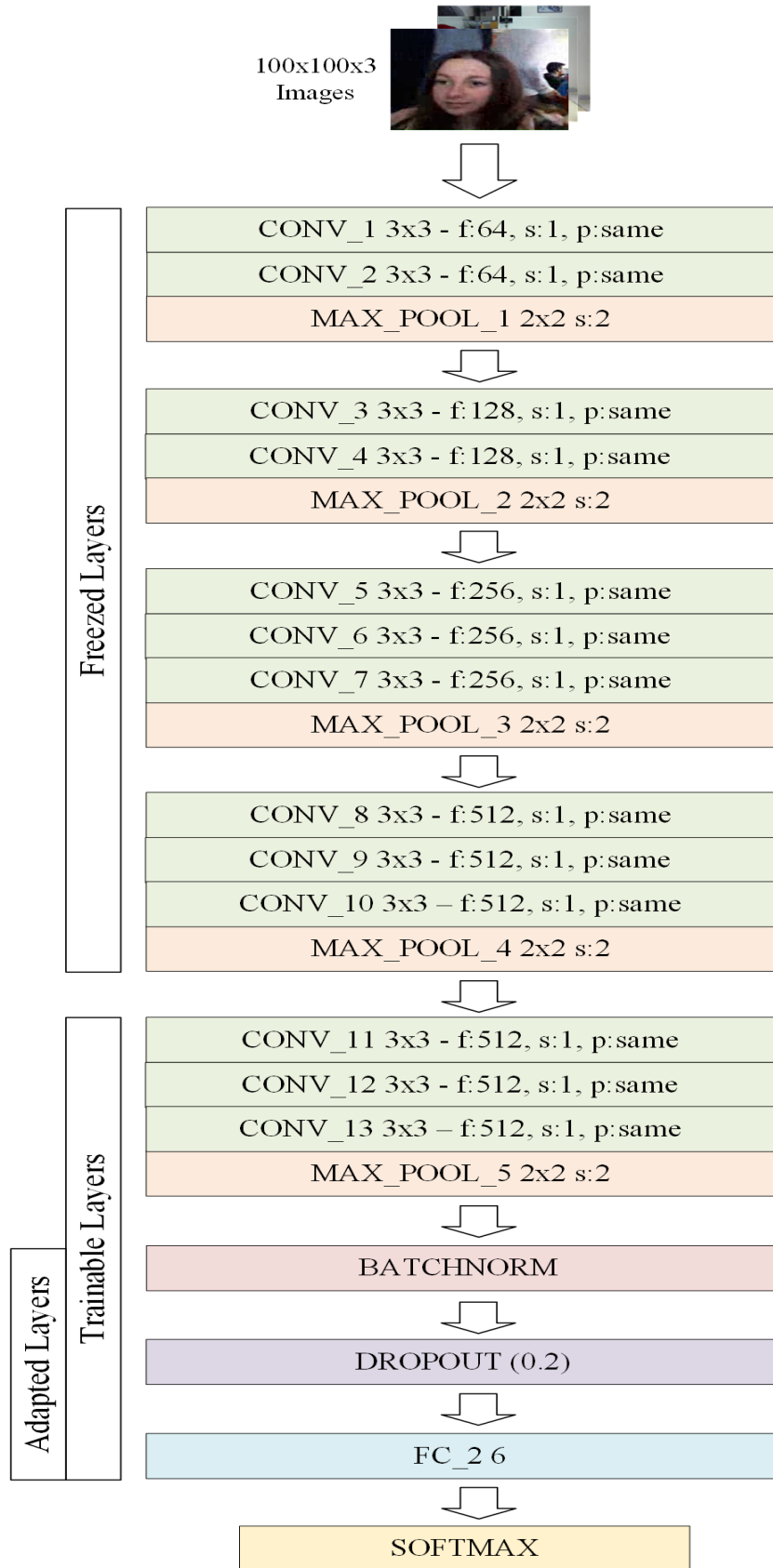


Figure 28: The adapted VGG16 network architecture

VGG16 – Trained on ImageNet	VGG16 – Last layers removed	VGG16 – New layers added
===== Total params: 138,357,544 Trainable params: 138,357,544 Non-trainable params: 0	===== Total params: 14,714,688 Trainable params: 14,714,688 Non-trainable params: 0	===== Total params: 14,719,814 Trainable params: 14,680,070 Non-trainable params: 39,744

Figure 29: The parameters of the pre-trained and adapted VGG16 network

3.3.2 Adapted Inception-v3 Network

Figure 30 shows the best-adapted version of the Inception-v3 network. In contrast to the adapted VGG16 network, all the layers from the beginning until the end (including the adapted) keep unfrozen. The network gets adapted by adding two fully connected layers after the last max-pooling layer, to transform the multidimensional result of the max-pooling operation in a 1-dimensional. The first fully connected layer consists of 1024 neurons and executes a non-linear a ReLU activation function and the second one is of size 6 and executes the SoftMax function. These two layers reduce the amount of calculation as explained in section 2.2.3. While testing different adaptations, using dropout and batch normalization either alone or in combination, the performance got deteriorated. Because of that reason these options were not further considered for the adaption. Figure 31 shows all the parameters of the pre-trained Inception-v3 network on ImageNet and the adapted version. As can be seen, the parameters of the pre-trained model are much lesser than the parameters of the VGG16 network. More precisely the parameters are twice as much for the VGG16 network. Nevertheless, the adapted version consists of some more parameters than the pre-trained on ImageNet one, which doesn't hurt the training time.

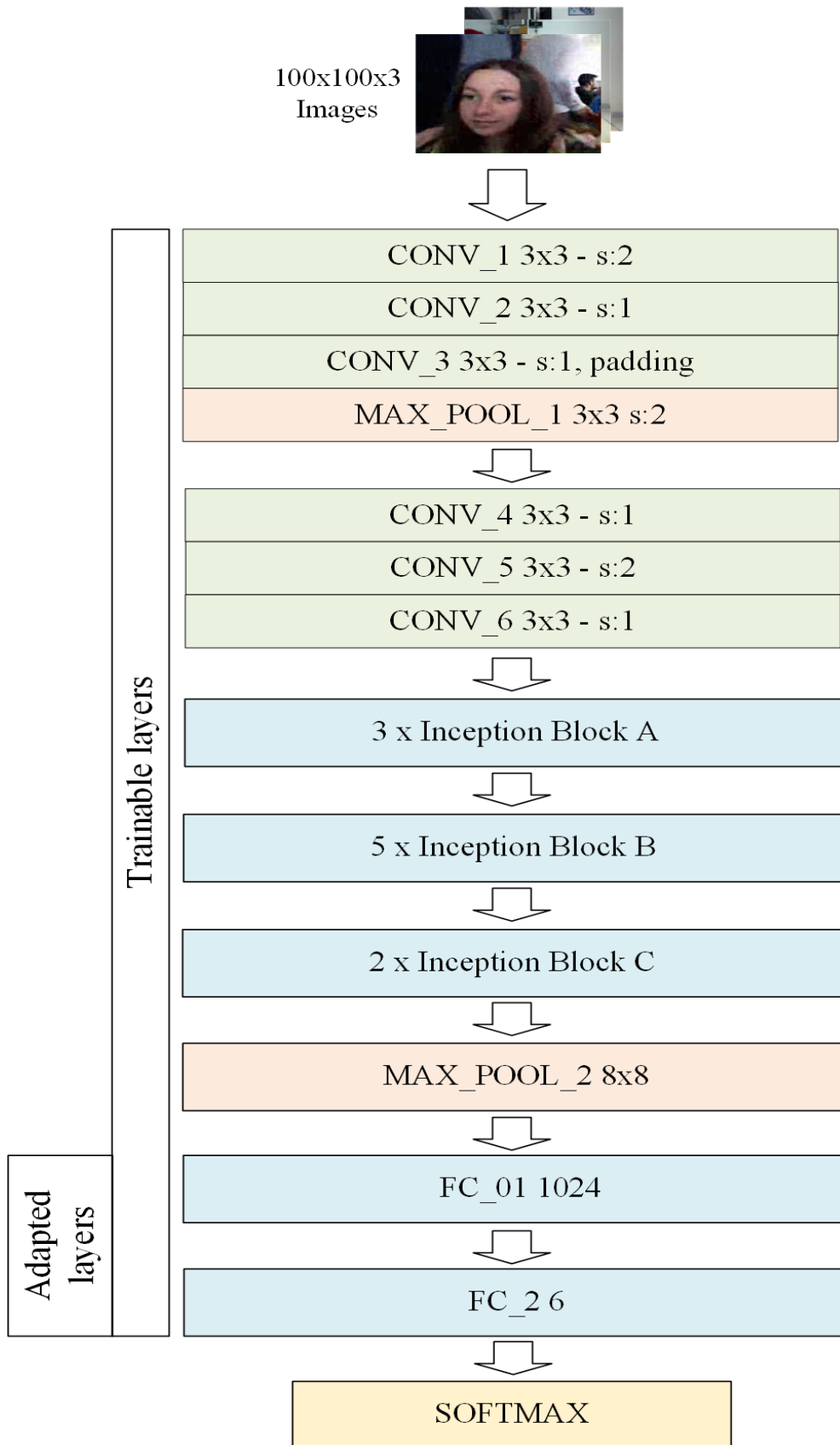


Figure 30: The adopted Inception-v3 network architecture

Inception-v3 – Trained on ImageNet	Inception-v3 – Last layers removed	Inception-v3 – New layers added
===== Total params: 23,851,784 Trainable params: 23,817,352 Non-trainable params: 34,432	===== Total params: 21,802,784 Trainable params: 21,768,352 Non-trainable params: 34,432	===== Total params: 23,907,110 Trainable params: 23,872,678 Non-trainable params: 34,432

Figure 31: The parameters of the pre-trained and adapted Inception-v3 network

3.3.3 Hyperparameter Setup and the Training Process

After adapting the pre-trained networks, it is time to set up all hyperparameters like the learning rate, the mini-batch size, and the epochs. It is recommended to use mini-batch sizes in a power of 2 i.e. 4, 16, 32, 64, 128, etc. because this speeds up in most cases the gradient descent algorithm. In this part, the optimizer has to be specified and the loss function to be chosen. The networks were trained initially using the following set up:

Hyperparameters

- Mini-batch size: 48, 64, 128
- Initial learning rate: 1e-3
- Max epochs: 100

Loss function

- Cross-entropy loss

Optimization algorithm

- Adam optimizer

At this point, the training is ready to start and to be executed in a cyclic manner. At the end of the training process, the performance gets plotted onto the screen, to specify the next steps based on this output to improve the performance. Figure 32 shows the execution of the training cycle.

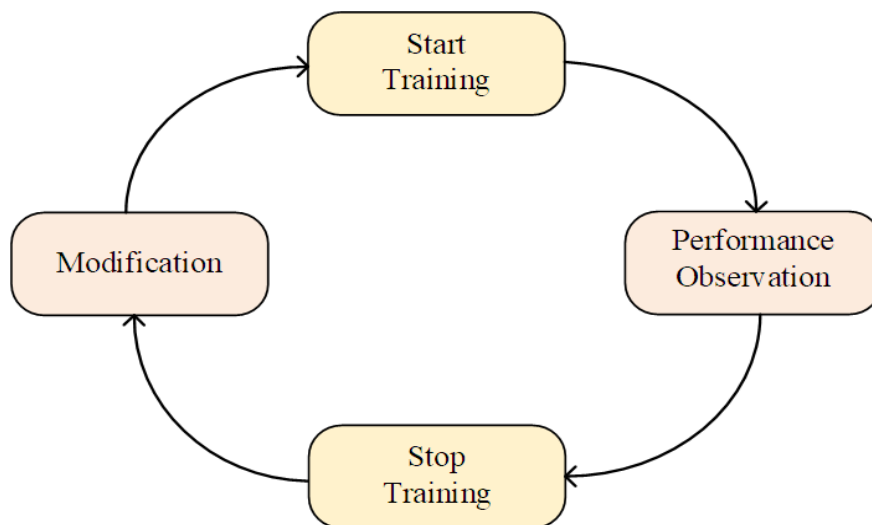


Figure 32: Training process

If the observation shows an unwanted behavior, it has to be stopped immediately, because as the progression of the training would only aggravate the error and unnecessarily drain the training time. In such a case, it is important to save the last checkpoint (in the project this is done via the corresponding callback) to restart after the changes from the last training point and not from scratch. The following listing illustrates the most common errors that can occur while training the network. In such a case, the training must be stopped immediately and changes made accordingly before the training is resumed:

- The training loss is much lower than the validation loss (overfitting)
- The training loss doesn't decrease after certain epochs (underfitting)
- The training accuracy doesn't improve after certain epochs (underfitting)
- The training accuracy gets 100% (overfitting)
- The validation loss after a certain point increase while the training loss decreases (overfitting)
- The validation accuracy is much higher than the training accuracy (underfitting)

A network performs properly during training time and when the training ends if the following observations can be made:

- The training and validation loss is decreasing
- The training and validation accuracy is increasing
- After a certain number of epochs, the training and validation loss gets nearly the same
- After the end of the training, the training accuracy keeps under 100%
- The training loss reaches the lowest value of 0.001 (gradients are nearly zero, training ends)

At this point, however, must be mentioned that it is completely normal that in the first epochs the validation accuracy achieves a low value and the validation loss a high one because the network starts learning at that point. In other words, the weights and biases are still not aligned appropriately, which changes while the training proceeds. If this value remains high and doesn't decrease significantly after a certain number of epochs, then the training has to be stopped immediately. In the next subsection, the underfitting and overfitting problems will be described in detail and possible solutions proposed to bring the network back to a stable state.

3.3.4 Underfitting and Overfitting

There are two terms that determine the quality of the training and validation process and, ultimately, the testing process. Underfitting occurs if the network doesn't fit the training data as Figure 33 left shows. This means that the network doesn't classify the training data appropriately and the error is very high. The opposite case which is called overfitting occurs, when the network fits the training data to 100%, but the validation data very bad. This is shown in the Figure on the right side. The disadvantage of that is that the generalization gets lost, which means the network fits only the training data, but on unseen data performs very bad. This would lead for example that an image that shows a dog gets classified to 90% as a horse, which is the worst result. The bottom of the Figure shows a good fit. The Figure shows that there are still outliers, but they remain less. Overfitting and underfitting are the most important problems a deep learning practitioner or researcher is facing.

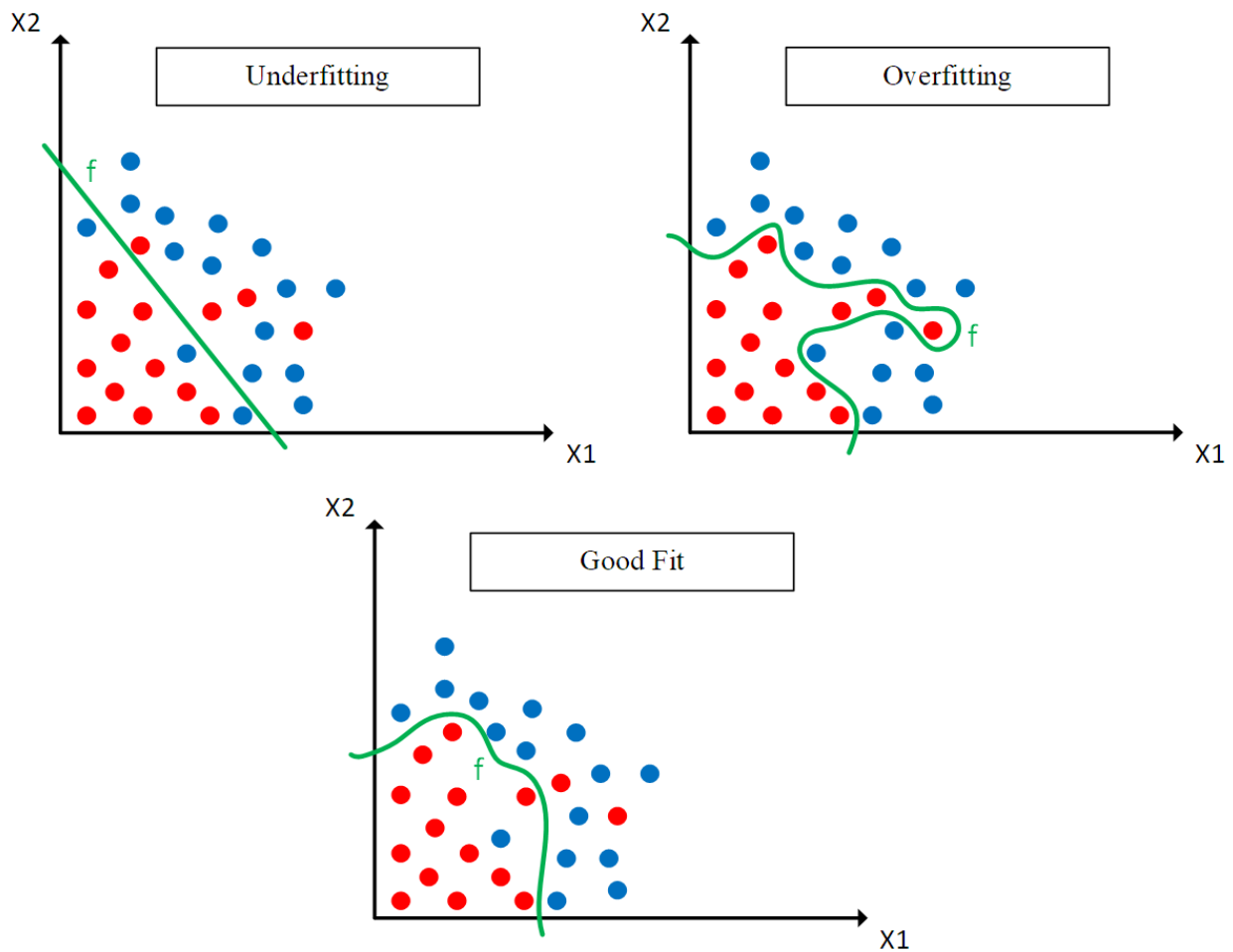


Figure 33: An example of underfitting and overfitting

To overcome the underfitting and overfitting problem, there are many solutions that can be applied like the following ones which get mostly applied in practice.

Underfitting

- Using a bigger network (i.e. more hidden layers, more neurons on a layer)
- Training the network longer
- Decreasing the number of input features
- Using another network architecture

Overfitting

- Using more training data (i.e. augmenting the dataset or use another bigger one)
- Regularization (i.e. learning rate decay, dropout, etc.)
- Increasing the number of input features
- Using another network architecture

The list shows that in the case of underfitting, a bigger network can be used, which means using either more layers or more filters on each layer. Increasing the training time may lead the network to stabilize after a number of epochs. Decreasing the number of input features excludes features which would make the training progress more difficult to learn from. If none of the solutions improves the performance, then it is recommended to switch to another network type. Overfitting occurs in most cases because the training set is too small. In that case, it is recommended to augment the dataset with more data or to use another bigger dataset. Data augmentation is the method that gets used in this work to create enough training data. If this doesn't help, applying different regularization techniques is the next common way to reduce overfitting. Available regularization techniques will be described in subsection 3.4. Increasing the number of input features (doing the opposite as in the underfitting case) can lead the network to learn better. The last option if none of the previous work is to switch to another network type. In addition to the above activities, it is also advisable to make changes to the hyperparameters to improve the performance of the network. This can include increasing or decreasing the learning rate, changing the mini-batch size, using another optimizer or increasing the epoch number. Figure 34 shows in a graphical manner the regions where underfitting and overfitting occurs and how the loss looks like in such a case. In addition, the orange dashed line shows the performance of a good fit. In most cases, at this point, early stopping gets applied, which ends the training immediately if the training performance increase while the validation performance decreases. On the left side of the graph, the training and validation losses are high, which means that the network underfits the data. In contrast on the right side, the training loss decreases while the validation loss increases. This concludes that the network underfits the data.

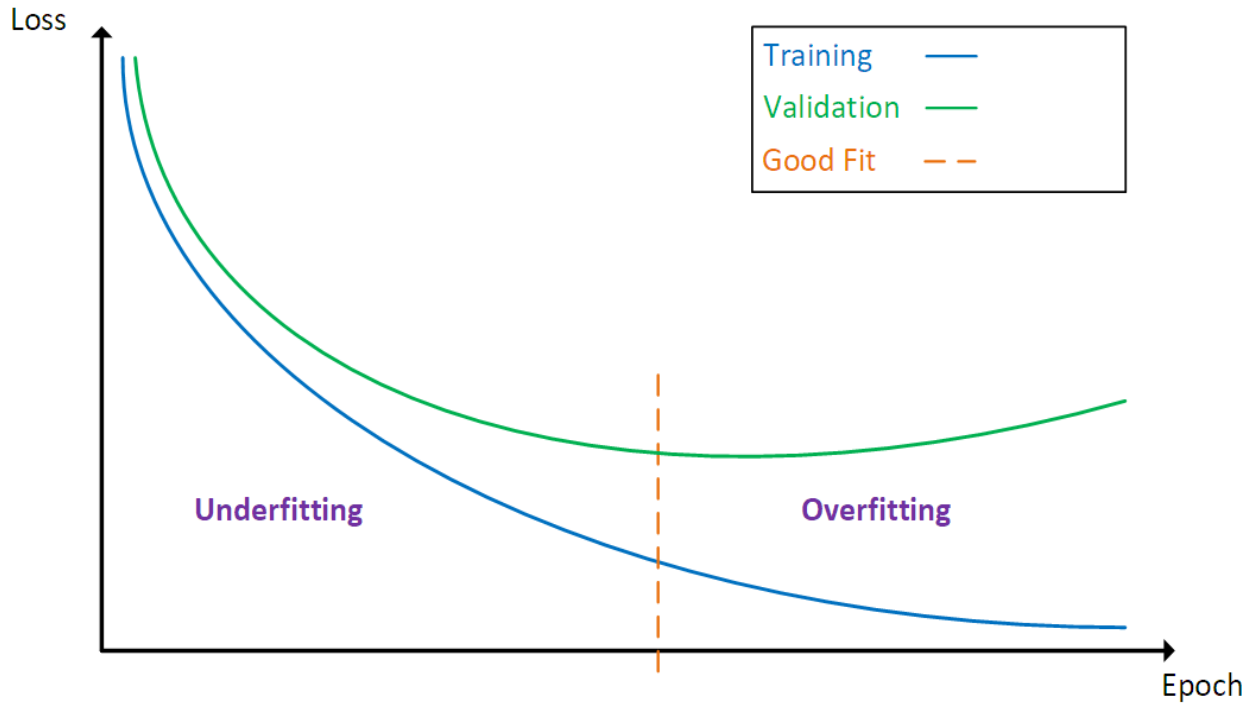


Figure 34: Underfitting, overfitting and good fit

3.4 Regularization

Regularization is a technique which prevents the network from overfitting during training time. This section describes some of the most used methods to apply these techniques.

3.4.1 Learning Rate Decay

One of the easiest regularization methods is to apply learning rate decay, which decreases the learning rate over time (steps). In the project, the learning rate gets reduced after each epoch. The formula to calculate the new learning rate based of the old one is the following, where the decay rate is the percentage of the learning rate to decay, α_0 is the initial learning rate, which is one of the hyperparameters set up at the beginning:

$$\alpha = \frac{\alpha_0}{1 + decay_rate \cdot epoch_num} \quad (5)$$

For example, if the current learning rate is 0.02, the decay rate is 0.1 and the current epoch is 10, then the new learning rate is 0.01. The learning rate decay helps the network to converge to a global minimum while the steps are getting smaller and smaller. Nevertheless, there is no guarantee that the network will converge to a global minimum even when learning rate decay gets used.

3.4.2 Dropout

Another neat regularization technique which contributes to reducing the overfitting problem is dropout [27]. This technique disables every iteration randomly a certain percentage number of neurons and achieves so a smaller network. This, in turn, reduces the overfitting problem, because the training data gets trained on a smaller network. The network size changes every iteration because different neurons are getting randomly disabled on each step. In general, it is recommended if a network is overfitting then the first choice is to use the dropout technique. In computer vision in most cases there are not much training data, so dropout prevents the networks here from overfitting quite well. It should be mentioned that the dropout operation gets disabled while validating and testing the network. In Figure 35 an example gets shown in applying dropout. In the left network, the first hidden layer drops out 25% of its neurons and the second hidden layer 50% of its

neurons. In the second network, only the second hidden layer drops out 50% of its neurons. This example shows that every iteration another network size gets trained.

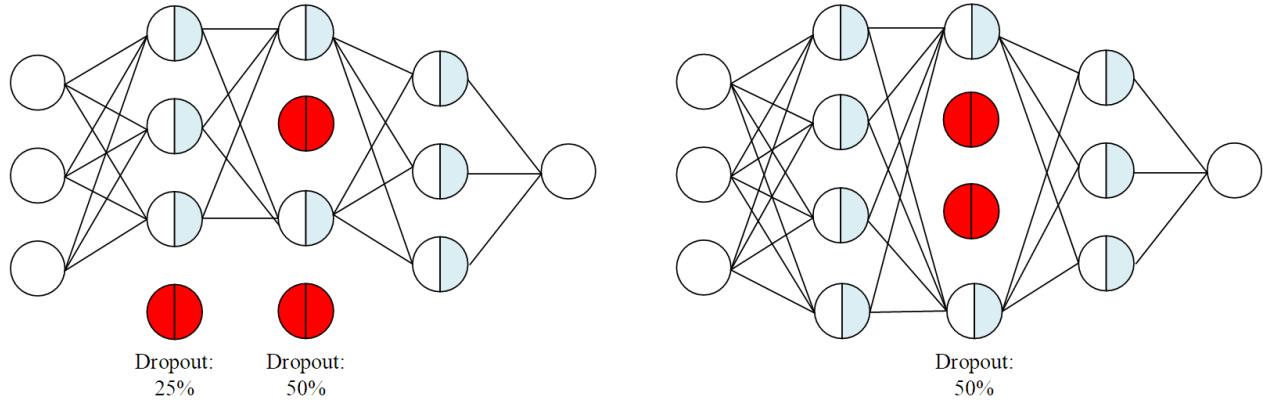


Figure 35: Examples of applying dropout

In general, dropout provides one of the easiest ways to reduce overfitting. Nevertheless, one downside of using dropout is that the loss function keeps not well defined because the loss changes every iteration based on the network size. In other words, the gradient descent algorithm is not able to work smoothly.

3.4.2 Batch Normalization

Batch normalization [28] is a technique which speeds up the learning process because the unnormalized input gets normalized into a specific range before the result proceeds to the activation function. This means that batch normalization zeros out the mean and makes the variance one. Figure 36 shows on which part batch normalization gets applied in a neuron. Normalizing means to convert all the input values so they lie all in the same range e.g. [-1, 1]. In deep learning the batch normalization gets applied before the fully connected layer, that is before the next activation function. This is necessary because otherwise, the unnormalized data would be processed by the activation function and not the normalized. In general, this technique makes the weights of deeper layers more robust to changes than in the first layers.

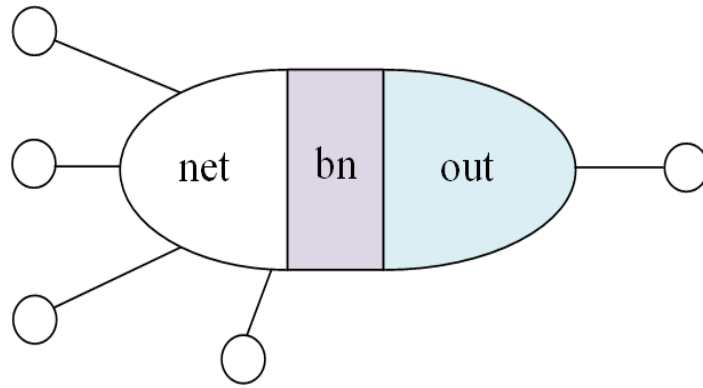


Figure 36: The intermediate process in a neuron in case of batch normalization

In the last step after regulating the network and doing some modifications on the hyperparameters, the training process can be resumed and observed again from that point. One important point that should be mentioned, is that it is recommended to store after every epoch the network weights, only if the validation accuracy increases in case an unwanted interruption occurs. In that case, if the network wouldn't be saved before, the overall training process had to be restarted again from scratch which increases the whole training time.

3.5 Evaluation

The performance of the the adapted and fine-tuned networks gets evaluated using on one side the test set of the FER2013 dataset and on the other side using 24 unseen images, which are provided as mentioned from the Dept. of Information and Communications Systems Engineering from the University of the Aegean. These images are of the type RGB and all of them are of different sizes. The faces are not aligned appropriately like the faces of the FER2013 test set, which means that each face occupies its own amount of space in the image. The whole evaluation process using the 24 unseen images is depicted in Figure 38. In the first step, all unseen images and the trained network to be examined are getting loaded from disk. While loading the images they get converted into grayscale images, because they are all of type RGB. This is not necessary but is equated with the training images that are of a grayscale type. The next step chooses the first image and pre-processes it appropriately. The image gets sharpened and resized to fit the dimensions of the network. It should be mentioned that the image remains 3 dimensional despite the fact that it is a grayscale one. If the image would be converted into 1-dimension, then the first layer(s) of the network had to be retrained from scratch to recognize them. In most cases, this is not a problem, but in general, while using a pre-trained network it is recommended to use images of input dimensions according to the pre-trained network. At next, the prediction process gets executed and each result of the predicted image gets printed on the screen like Figure 37 shows. After all, images are predicted a bar chart gets plotted which shows the distribution of the emotion classes. The results of both evaluation processes will be discussed in section 4.

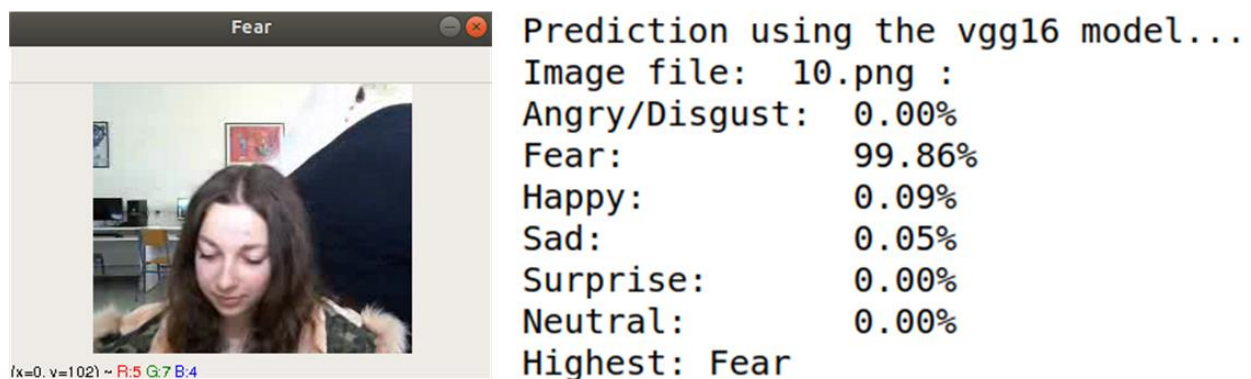


Figure 37: An example of prediction using an unseen image

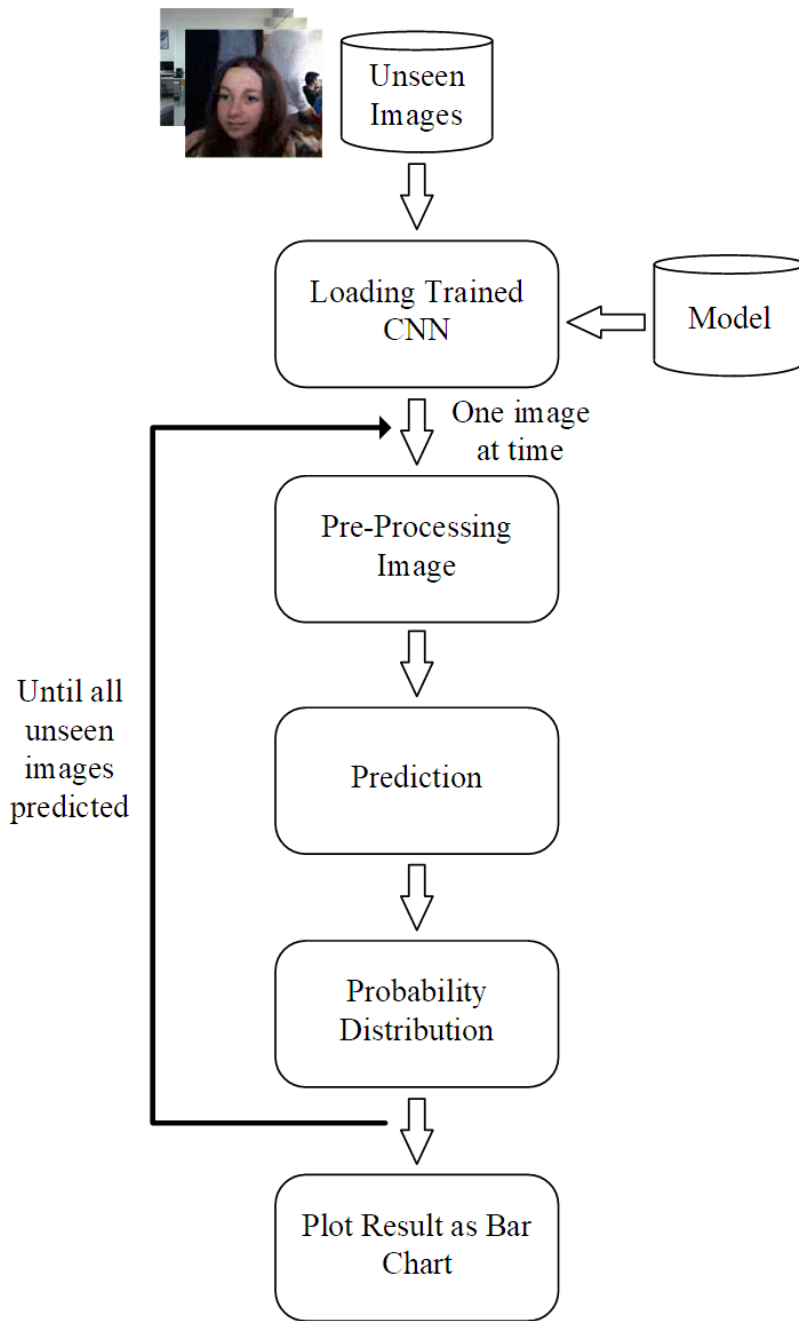


Figure 38: The prediction process using unseen images

4. Experimental Results

This section evaluates the training and test process of the adapted and fine-tuned networks VGG16 and Inception-v3. To achieve a suitable network structure a lot of experiments were made based on the structure of the respective paper. One of the first experiments to do transfer learning is to replace the prediction layer (last fully connected layer) with another one, which handles only six instead of 1000 classes. In addition, many adaption variations can be tried, to find the best network structure for solving the FER task. Nevertheless, it is not as easy to say that one solution is the best one, because there are plenty of others, which give similar or even better results. The following list shows some variations which were tried:

- Adding multiple fully connected layers
- Applying dropout with different dropout parameters i.e. 0.1, ...,0.9
- Adding multiple dropouts
- Adding Batch normalization
- Freezing and unfreezing one or multiple layers in the adapted part

To illustrate how the networks performed during the training and the validation process, the progress of the loss, accuracy and the learning rate were plotted after the training ended. The training ended either after the specified epochs were reached, or if the validation accuracy didn't improve anymore (early stopping). For the sake of the prediction quality of the validation, a confusion matrix was plotted as last plotted. Figure 39 shows an example of how a confusion matrix looks like. Each row represents the predicted labels and each column represents the target ones. The values in the fields represent the number of the right and wrong predicted emotions. For example, the Figure shows that in the first row and second column, the target label Fear was predicted as Angry, Disgust, which is obviously wrong. Only the values on the diagonal are the labels which were predicted right. For example, row three and column three shows the right predicted label Happy. All values that are outside the diagonal are wrong predictions. The color of the confusion matrix shows the number of predictions of the respective field. The color starts from a brighter color which shows a less value and ends in a dark color which depicts the highest value. For example, the class Angry, Disgust has the most predicted number. The colored representation allows a quick overview with the eye.

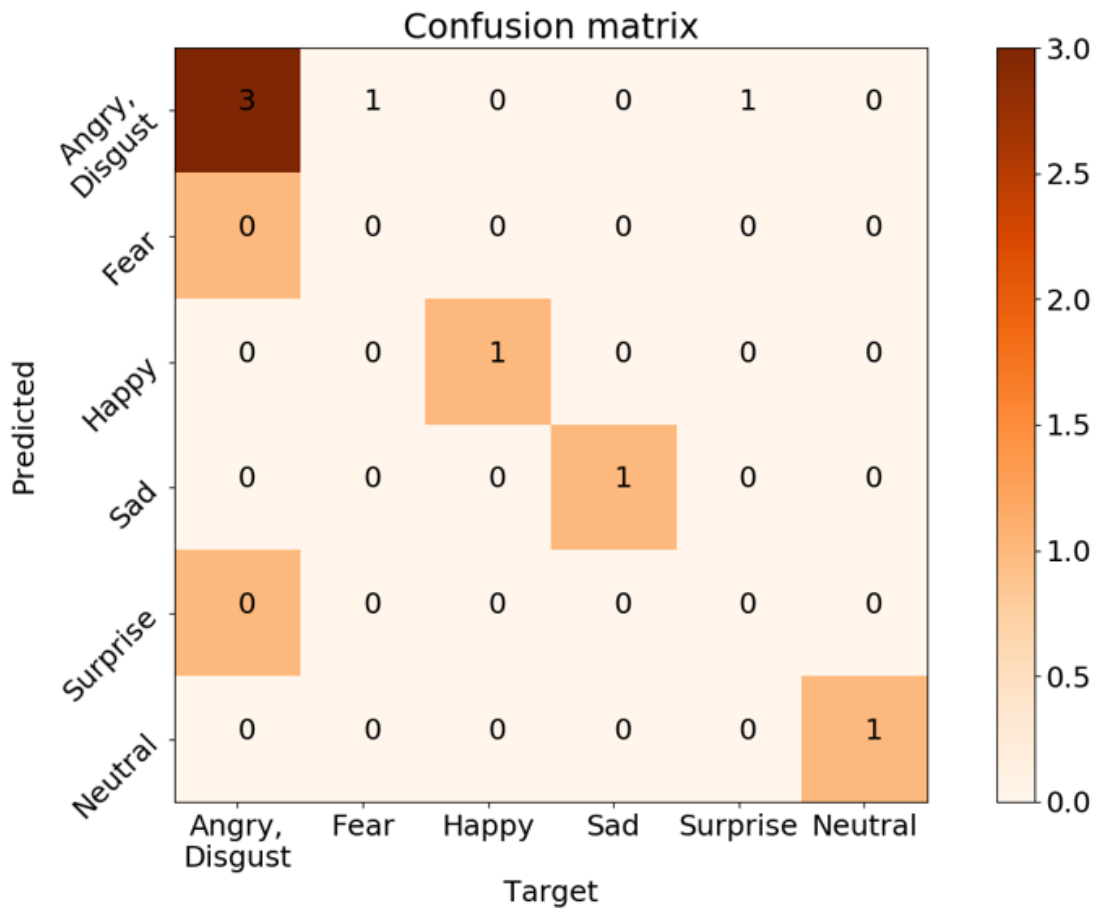


Figure 39: An example of a confusion matrix

To test the performance of the networks, the test set of the FER2013 dataset and 24 unseen RGB images were used, provided by the Dept. of Information and Communications Systems Engineering of the University of the Aegean as mentioned earlier. Because there were no target labels available for the unseen images, the prediction could only be evaluated using a bar chart, which shows the number of images predicted based on the emotions. Most of the images expressed the emotions Fear and Sad because they contained students who were writing exams and their mood was accordingly. The overall training and prediction process got executed 80% using the GPU and 20% using the CPU. Nevertheless, the memory allow_growth flag in the code was set to True, to allow using more GPU memory if it was required. In general, as mentioned before to train very deep neural networks, powerful computer systems are required either locally or using a cloud-based system. In section 4.3 the results of both networks will be compared to each other and to the solutions mentioned in section 2.3. The following Table 5 contains the performance of the networks achieved at the ILSVRC competition [25].

Network type	Top-1 error rate (%)	Top-5 error rate (%)	Top-1 Accuracy
VGG16	24,7	7,3	71%
Inception-v3	21,2%	5,6	77,9%

Table 5: Top-1 and Top-5 error rates of CNN's

The table shows that at the competition the Inception-v3 network outperformed the performance of the VGG16 network in all cases. This is expected due to technological improvement as opposed to the older ones. However, it doesn't mean that a network which performs on one specific task like object detection does necessarily perform the same as well on a different task like facial emotion recognition. The same applies to the use of a particular record for a particular network over another. If the adapted and fine-tuned Inception-v3 network outperforms the adapted and fine-tuned VGG16 network for the FER task, will be examined in section 4.3. Nevertheless, the VGG16 networks are still widely used for both research and commercial applications. The reason is that they do not need so many resources compared to newer networks and can, therefore, be trained on conventional computer systems, which don't provide too much computational power. In addition, it is more preferable to lose some performance and save computer resources. Furthermore, the VGG16 networks still deliver excellent results in computer vision applications like object detection and object recognition.

4.1 Experimental Results of the Fine-tuned VGG16 Network

This section evaluates the finally adapted and fine-tuned VGG16 network shown in Figure 28. As proposed in section 3.3.1 the last layers of the network got adapted appropriately to solve the FER task. In order to achieve a rational structure, as mentioned above, many different experiments have been made. The goal was to achieve a similar or even better performance nearly to the performance of 71%, which is listed in Table 5.

4.1.1 First Experiments

While trying different adaptations, it could be observed that without applying learning rate decay, the network started to overfitting in the first epochs. Trying different hyperparameters, unfreezing more layers and adding regularization techniques mentioned in section 3.4, didn't fix that problem. The only difference was that the network started overfitting later. After applying learning rate decay with a decay rate of 50% and reducing the learning rate to $1e-4$, the overfitting problem couldn't be observed anymore. In the first experiments, all the layers except the adapted ones kept frozen, which caused the network to underfit for a lot of epochs (validation accuracy was higher than training accuracy and validation loss was greater than training loss). Freezing backward more hidden layers and keep all the others trainable, caused the network to achieve a training accuracy of 71% and a validation accuracy of 65% as Figure 40 shows. The learning rate as can be seen decreased smoothly based on the formula (5) of section 3.4.1. When the learning rate reached a very low value in the 4th epoch, the accuracies progressed as expected. It could also be observed that the accuracies increased smoothly with decreasing learning rate. After the 24th epoch the training ended, because no improvement of the validation accuracy could be observed. The maximum achieved accuracy of the network was 71% training and 65% validation accuracy. Since the difference between the validation accuracy achieved and the accuracy of Table 5 was still 6%, the adaption and fine-tuning process continued until the difference was as small as possible. Figure 41 shows the confusion matrix of the validation for the described experiment. The matrix shows that the network correctly predicted most of the values for all emotions because the diagonal contains the most values of each class. Based on this observation, it could be concluded that the network was learning right. Nevertheless, because of the big difference of the accuracy mentioned above the adjustment had to proceed to achieve a better result.

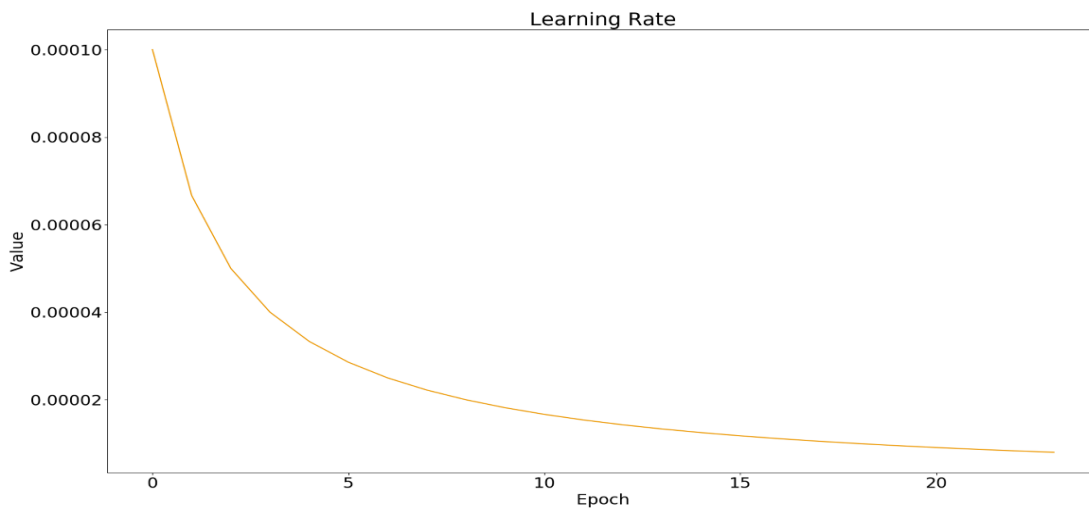


Figure 40: Performance measure of the VGG16 network for the experiment described [61]

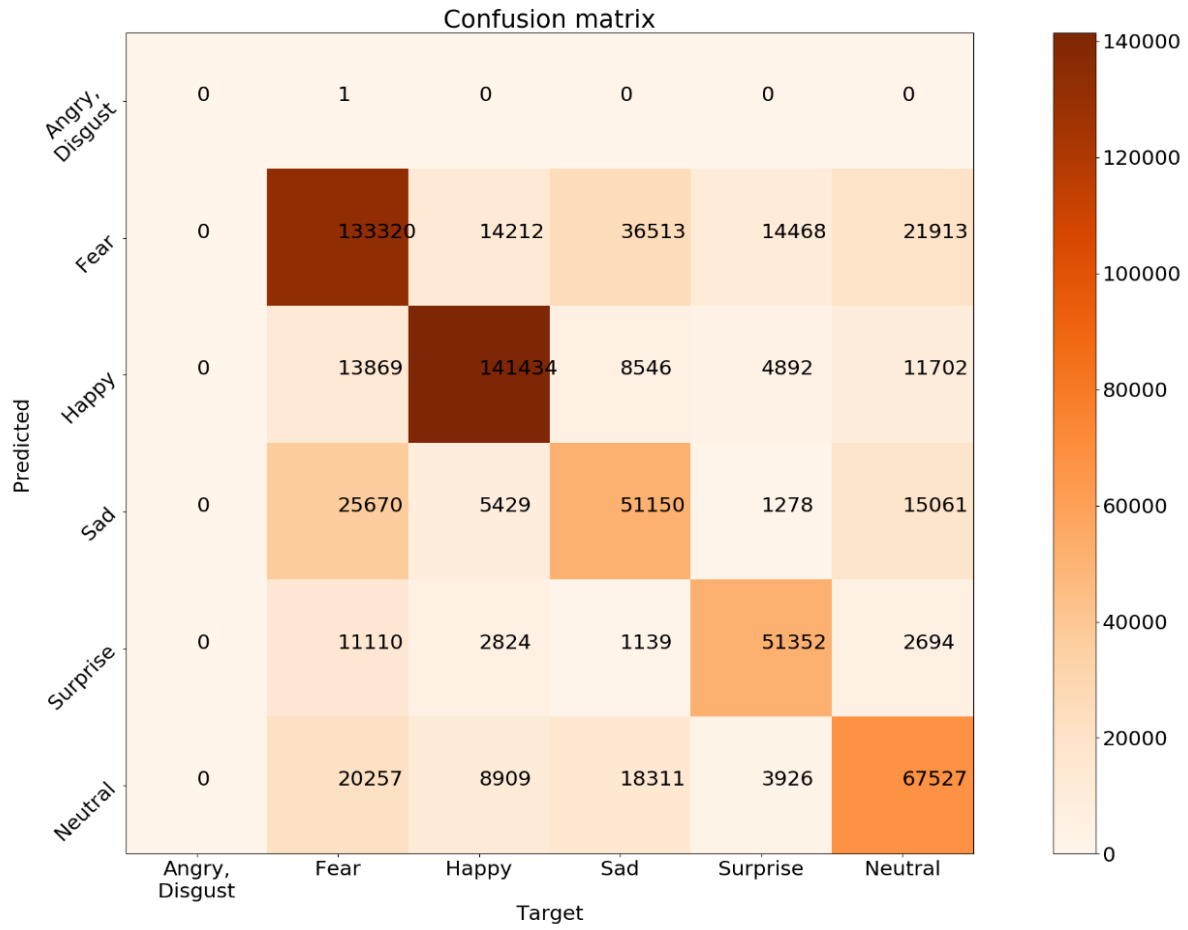


Figure 41: Confusion matrix of the validation for the VGG16 network for the experiment described

4.1.2 Best achieved Training Result

While training the network using the adapted version of Figure 28, the performance got improved significantly. As the Figure shows only the first two convolutional layers kept froze and all the following layers were set to trainable. The reason to freeze only the first layers was that the pre-trained network weights of the filters were still adjusted learn simple features like edges, curves, etc. and so they didn't need to be re-trained again. For the training process, the same initial hyperparameters were set up. Figure 42 shows the difference to the previous performance by freezing only the described layers. As a result, the accuracies started to improve quicker from the 3rd epoch instead of the 5th as before. The network achieved in the 16th epoch instead as before in the 24th a maximum training accuracy of 74% and a validation accuracy of 71%, which was 6% higher than before. This concluded, that the network performance and learning process got improved in contrast to the previous experiments. It could also be observed, that the training and validation loss started to decrease earlier and the difference between them got smaller as the Figure shows. The same was true for the accuracy. The validation accuracy achieved was the same as the accuracy of the competition shown in Table 5. Figure 43 shows the confusion matrix of the validation for the best-achieved result. The matrix shows that the network correctly predicted more emotions than before because the outliers, as can be seen, got reduced. This concluded, that the network was better learning as before and so this network was used to get tested on the unseen images, which will be described next.

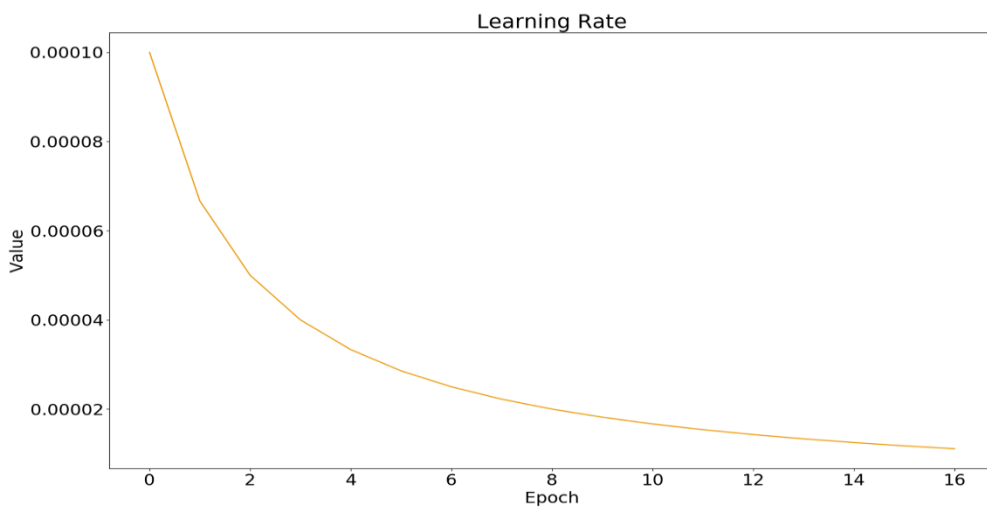
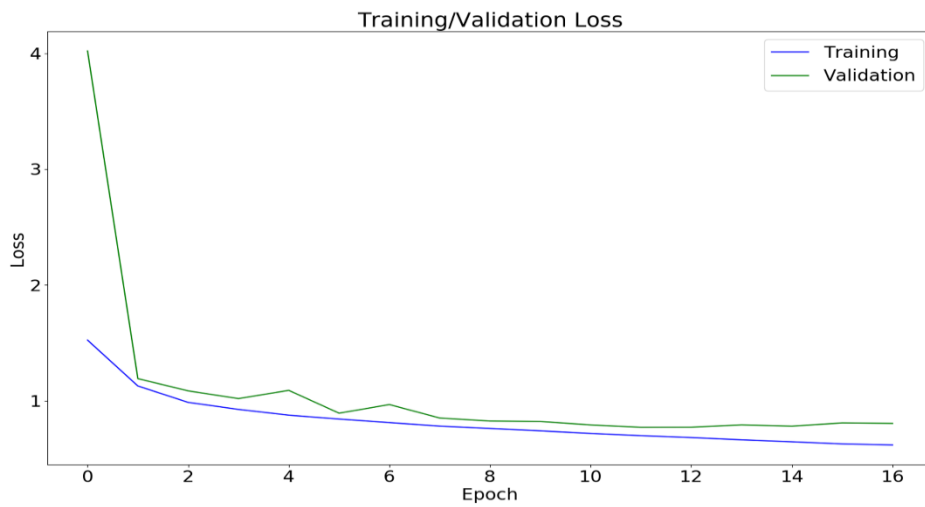
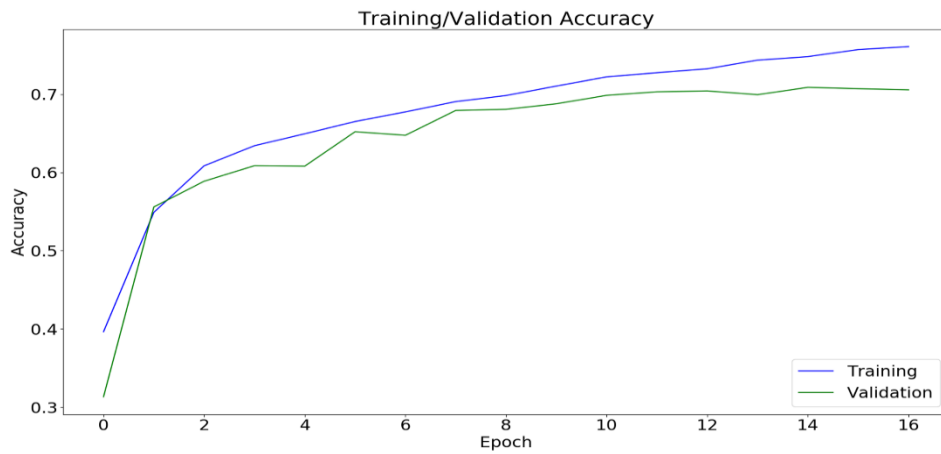


Figure 42: Performance measure of the best-achieved result for the fine-tuned VGG16 network

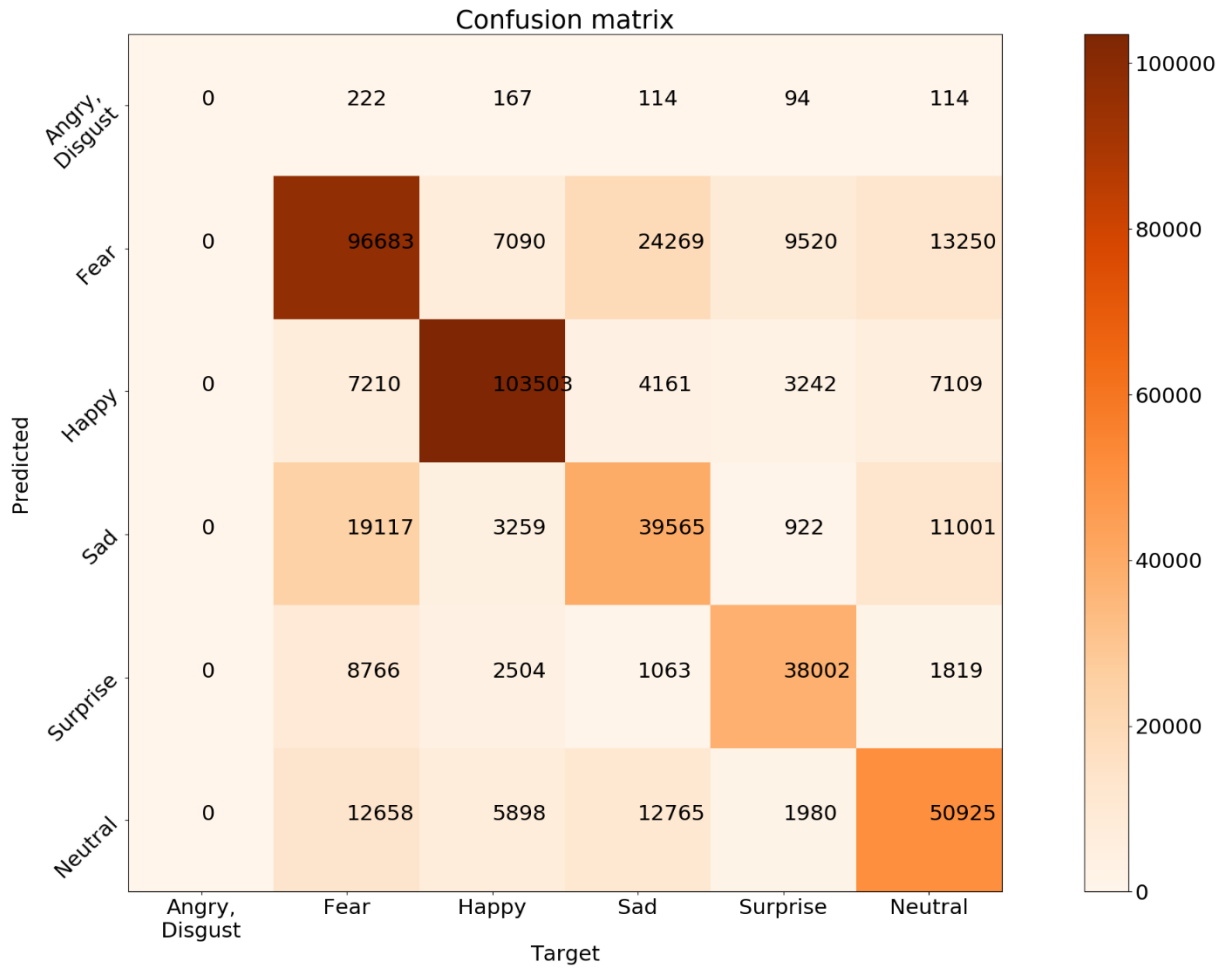


Figure 43: Confusion matrix of the validation for the best-achieved result for the fine-tuned VGG16 network

4.1.3 Result of the Evaluation

To evaluate how the network performed on the 24 unseen images, each image was considered separately. Unfortunately, because of the missing labels, the evaluation of this method could only be done based on the predicted percentage of each emotion and the perception of the tester. Figure 44 shows an example of the output for some images that got predicted. The bottom-right image, for example, shows that the emotion Happy was predicted with 54,51% as Happy, which is based on the human perception right. Another example is the top-right image which shows that the emotion Fear was predicted with 57,32%, which is indeed right, too. Nevertheless, there were images who expressed other emotions as they were predicted. An example is shown in the bottom-center image, which was predicted with 68,27% as Fear, but the human perception would conclude that this is a Happy emotion. To get an overview of all the predicted images, at the end a bar chart was plotted. Figure 45 shows the number per emotion class predicted based on the six emotion classes. The graph shows that the emotion Fear got the most predicted values (17/24), which is right because most students while writing an exam were frightened. The emotion Surprise got the second most values, which concludes that some students were surprised (maybe they expected to get easier or more difficult questions). The last most predicted emotion was the emotion Happy, which concluded that some students got the questions they were expecting. The result of the bar graph showed that the network had learned to classify most emotions accordingly, which makes it suitable for such tasks. The comparison of both evaluation methods using the adapted VGG16 network and the adapted Inception-v3 network, will be discussed latter.

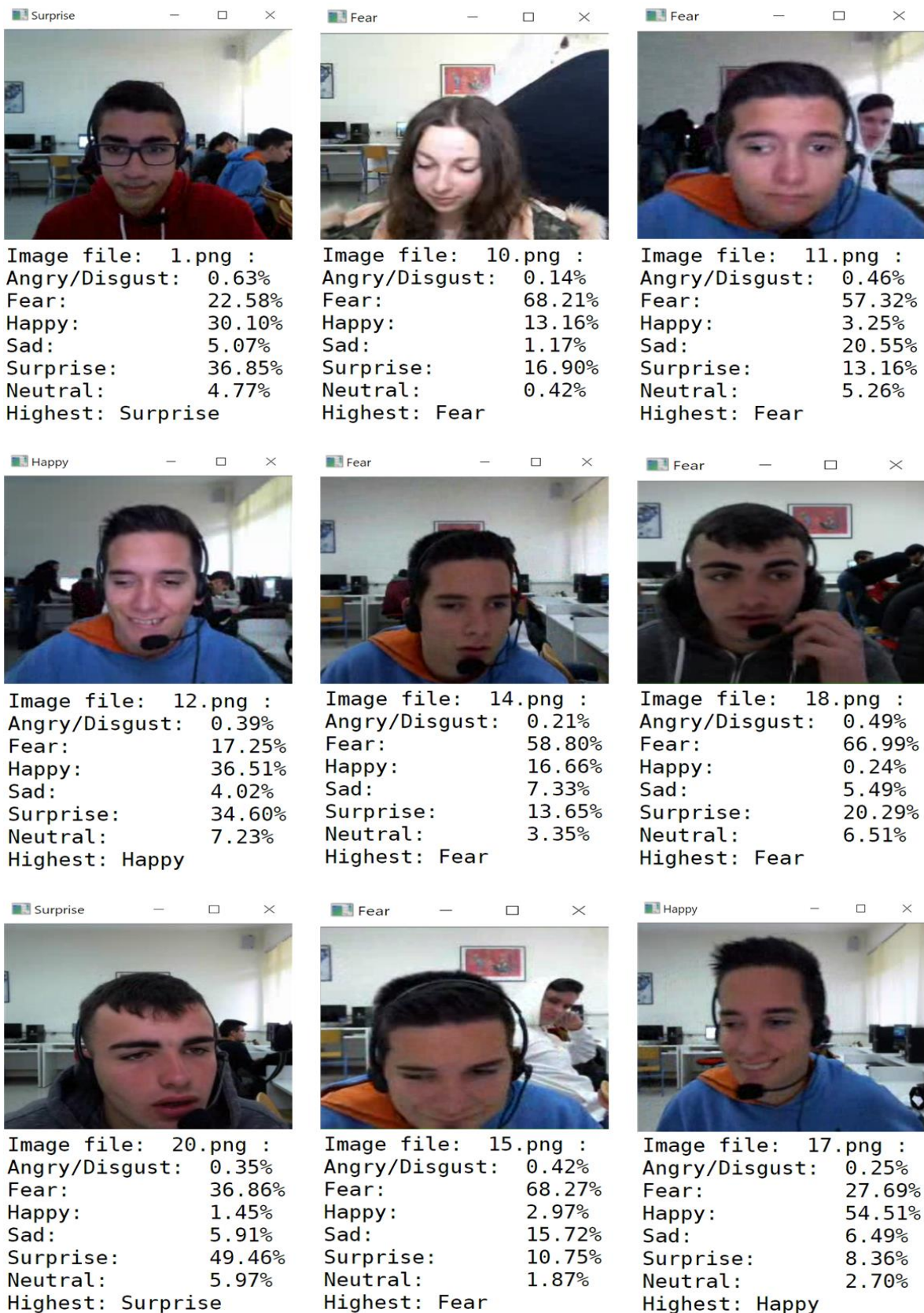


Figure 44: Output of the predicted unseen images for the fine-tuned VGG16 network

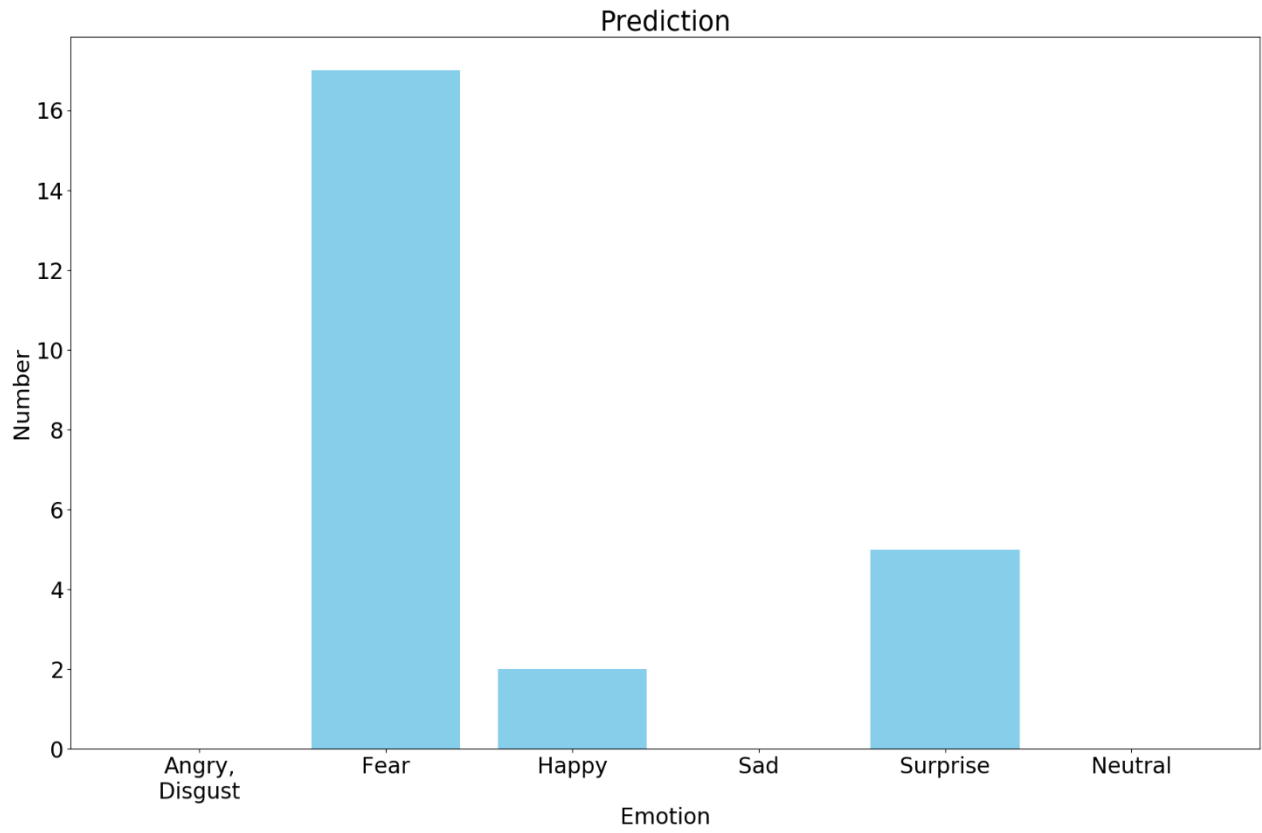


Figure 45: Prediction result of the unseen images for the fine-tuned VGG16 network

4.2 Experimental Results of the Fine-tuned Inception-v3 Network

In this section, the finally adapted and fine-tuned Inception-v3 network of Figure 30 will be evaluated how it performed to solve the FER task. Just like for the VGG16 network, the last prediction layer was adapted appropriately. The goal was to achieve a similar or even better performance (77,9%) based on Table 5.

4.2.1 First Experiments

In the beginning, the network was adapted as suggested at the beginning of the section. The hyperparameters were set up as before and the weights of the first until the last max-pooling layer got frozen. After starting to train the network it could be observed that after the 17th epoch the network achieved a training accuracy of 62% and a validation accuracy of 56%, which concluded that the network was overfitting. Adding dropout with a rate of 50% and batch normalization the training and validation accuracy deteriorated by 23,3% and 26,3% respectively. This is not unusual, because adding multiple regularizations techniques in most cases deteriorates the performance. Keeping only dropout with a rate of 20%, setting the learning rate to 1e-3 and freezing only the first convolutional layer, the training accuracy approached 66% and the validation accuracy 64% as can be seen in Figure 46. The Figure shows that the accuracies increased smoothly, but the validation accuracy and validation loss were oscillating. The learning rate reduction as can be seen contributed to the improvement of the performance. Figure 47 shows the confusion matrix of the validation for the described experiment. The matrix shows that the network didn't correctly predict most of the values for all emotions, because not all values on the diagonal are the biggest ones for each class. For example, the emotion Sad has the most outliers in contrast to the other emotions. Based on this observation, it could be concluded that the network still had problems to classify specific emotions accordingly and the difference of the accuracy to the Table 5 was r.a. 14%, which was too big.

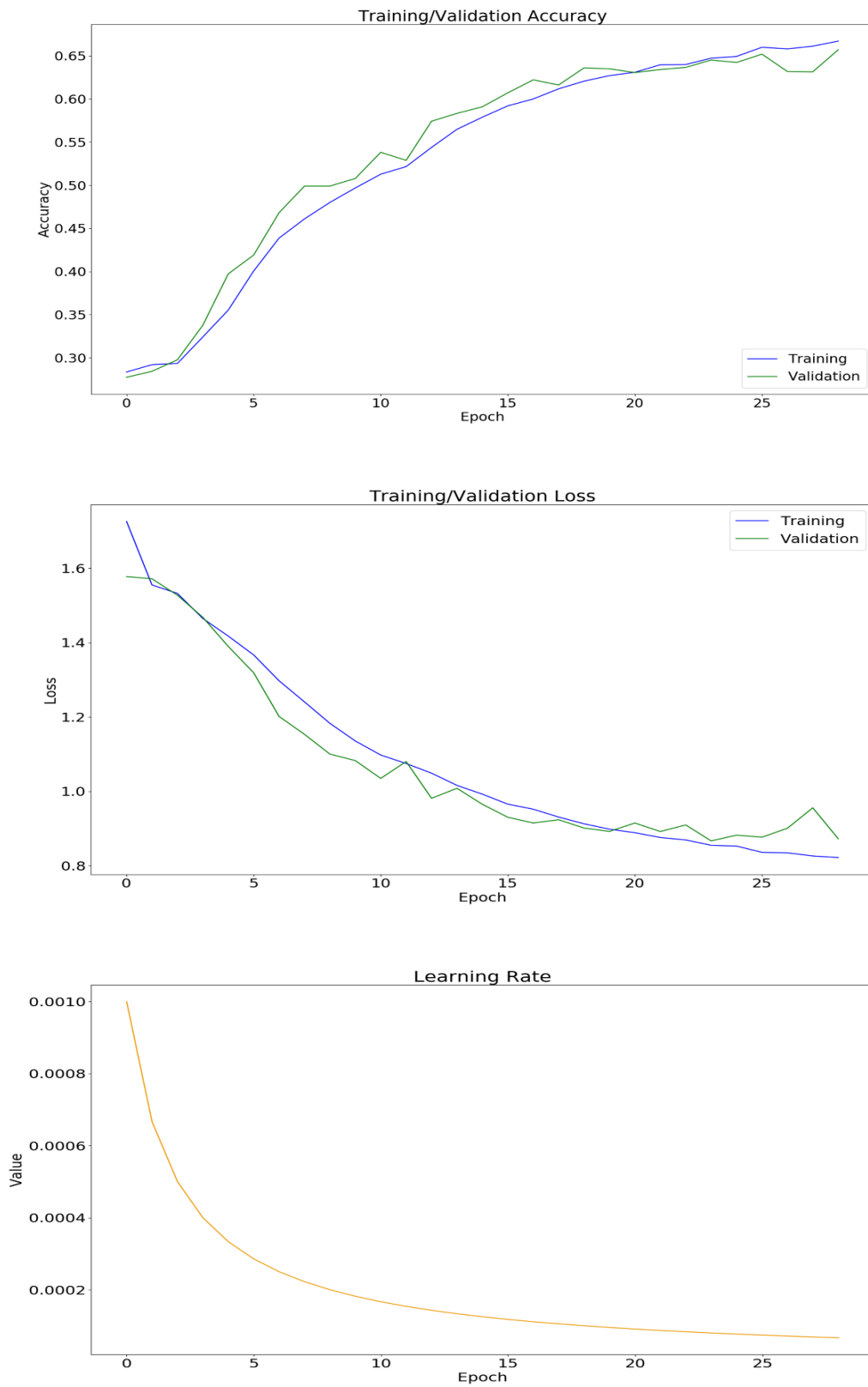


Figure 46: Performance measure of the Inception-v3 network for the experiment described

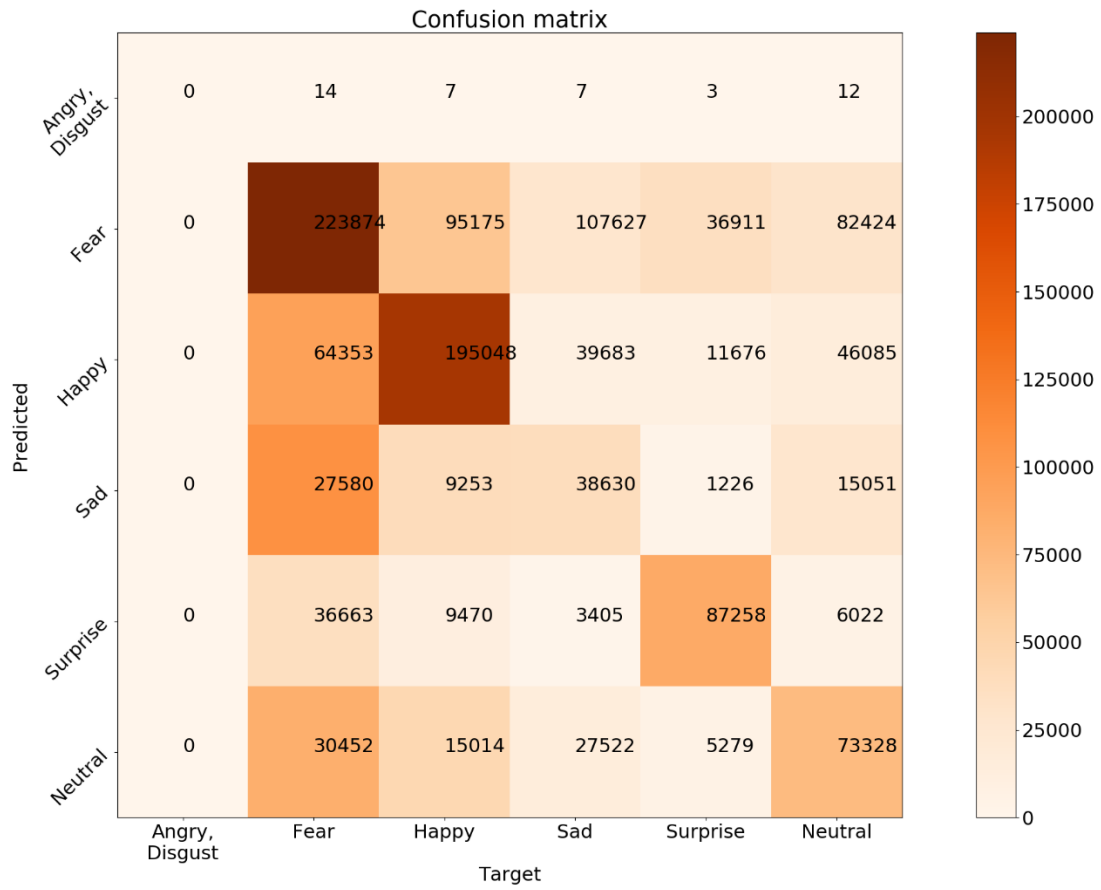


Figure 47: Confusion matrix of the validation for the Inception-v3 network for the experiment described

4.2.2 Best achieved Training Result

After using the adapted version of Figure 30 the performance got much better than before. This version unfroze all the layers and didn't use any regularization methods. Additionally, two fully connected layers of different sizes were added after the last max-pooling layer. The first one consisted of 1024 neurons and executed the ReLU activation function and the second one consisted of the six classes and the SoftMax function. After executing the training with the same initial hyperparameters as before, it could be observed in contrast to the previous experiment, that the accuracy and loss didn't oscillate like the previous one as can be seen in Figure 48. The network was underfitting the data until the 10th epoch, which wasn't a problem because one solution is to train longer. This changed after the learning rate had reached a lower value and proceeded smoothly while the learning rate kept decreasing. The network achieved after 22 epochs a training accuracy of 71,6% and a validation accuracy of 68%. This implies that the performance was improved by 4% compared to the previous experiment, but still differed by 10% from the performance of Table 5. Despite the further experiments were made, this could not be improved, in contrast, it got even worse. Thus, this solution was preferred for the subsequent test process. Figure 49 shows the confusion matrix of the validation for the best-achieved result. The matrix shows that the network correctly predicted more emotions than before, and the outliers got reduced significantly. As the Figure shows the emotion Sad got recognized, too. This concluded, that the network had improved its learning capability in contrast to the previous experiment, so this network was used to get tested on the unseen images, which will be described next.

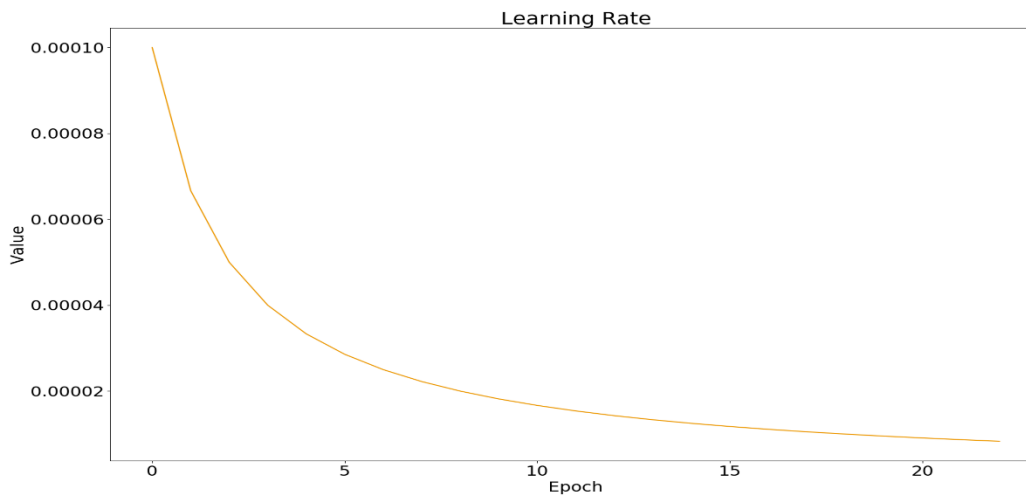
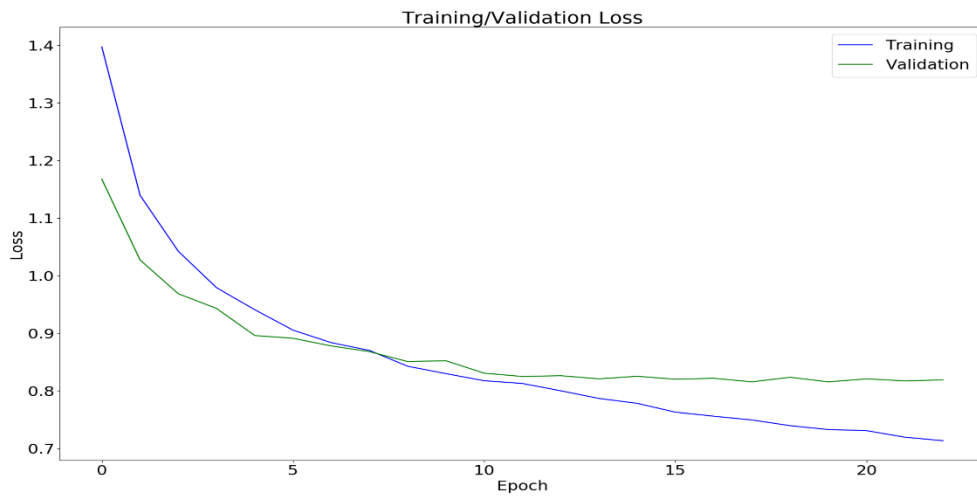
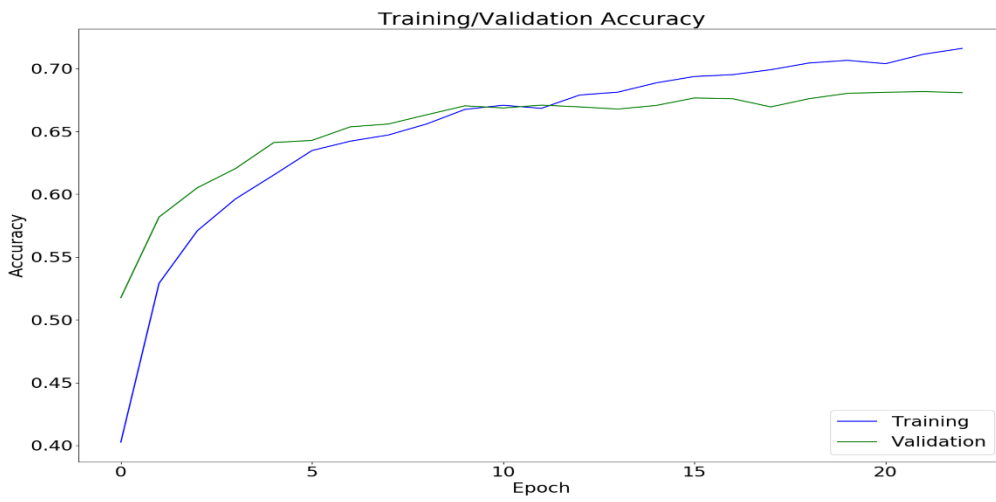


Figure 48: Best measured performance of the Inception-v3 network

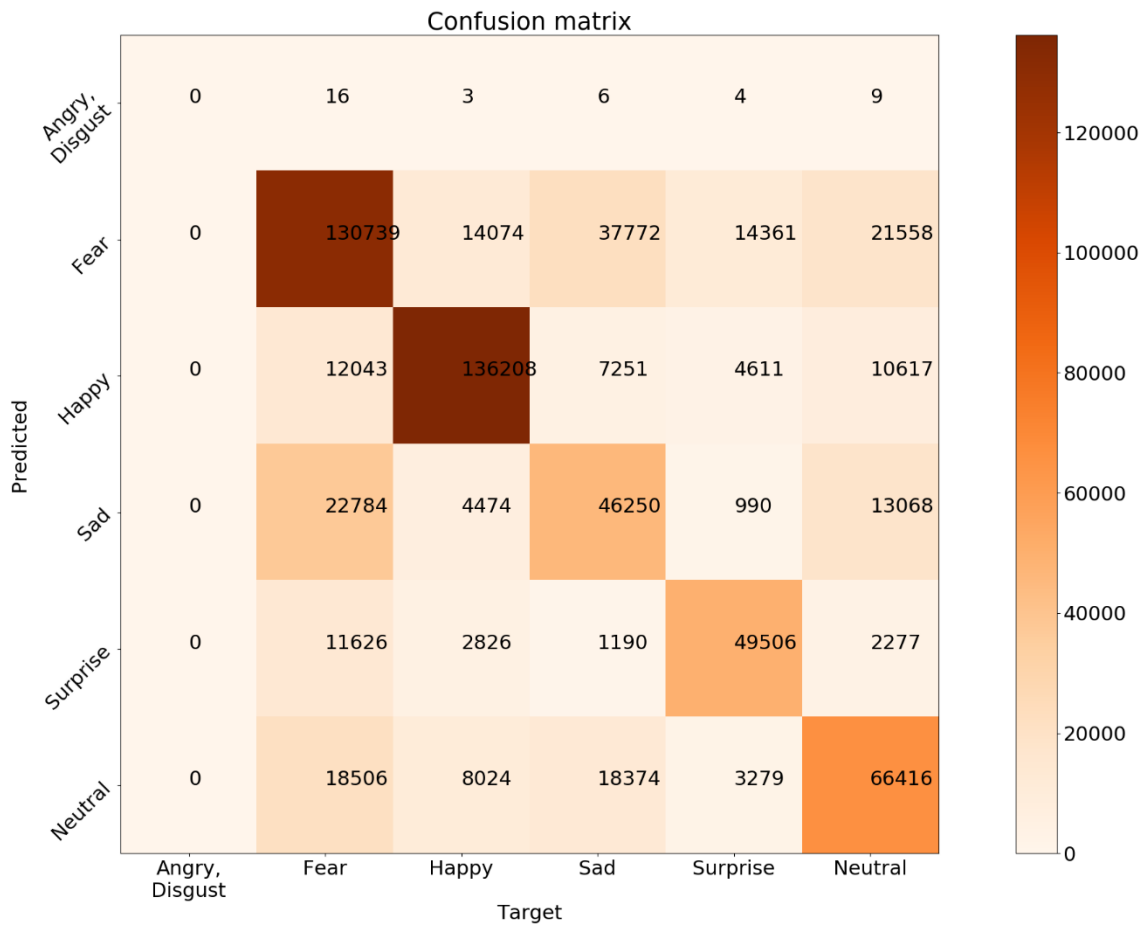


Figure 49: Confusion matrix of the validation for the best-achieved result using the fine-tuned Inception-v3 network

4.2.3 Result of the Evaluation

To evaluate the network, the same evaluation methods were applied like for the fine-tuned VGG16 network. Figure 50 shows the predicted output of the same nine unseen images as before. As the Figure shows for example in the second row the first image got misclassified as Fear, but the image showed an emotion of Happy. Nevertheless, the output shows that next predicted emotion was the right one with 18,39%, but the difference to the highest one was more than 43%. As can be seen, most of the pictures were classified as Fear. To get an overview of all the predicted images, at the end a bar chart was plotted as described before. Figure 51 shows that the emotion Fear got the most predicted values (21/24), which is not wrong because as mentioned earlier the images show students writing an exam and are mostly frightened. Nevertheless, the above example of the wrong predicted Happy emotion as Fear, shows that the network was miss-predicting this emotion (overall 2/24). Only one emotion was classified as neutral, which concluded that some students were neither anxious nor happy about the questions of the exam. Indeed, there were some images expressing this emotion. The result of the bar graph showed that the network had learned to classify most emotions correctly, but still made some important misclassifications. Nevertheless, the network is like the VGG16 suitable to solve FER tasks. The differences of the performances between both of these networks will be examined in the next section.

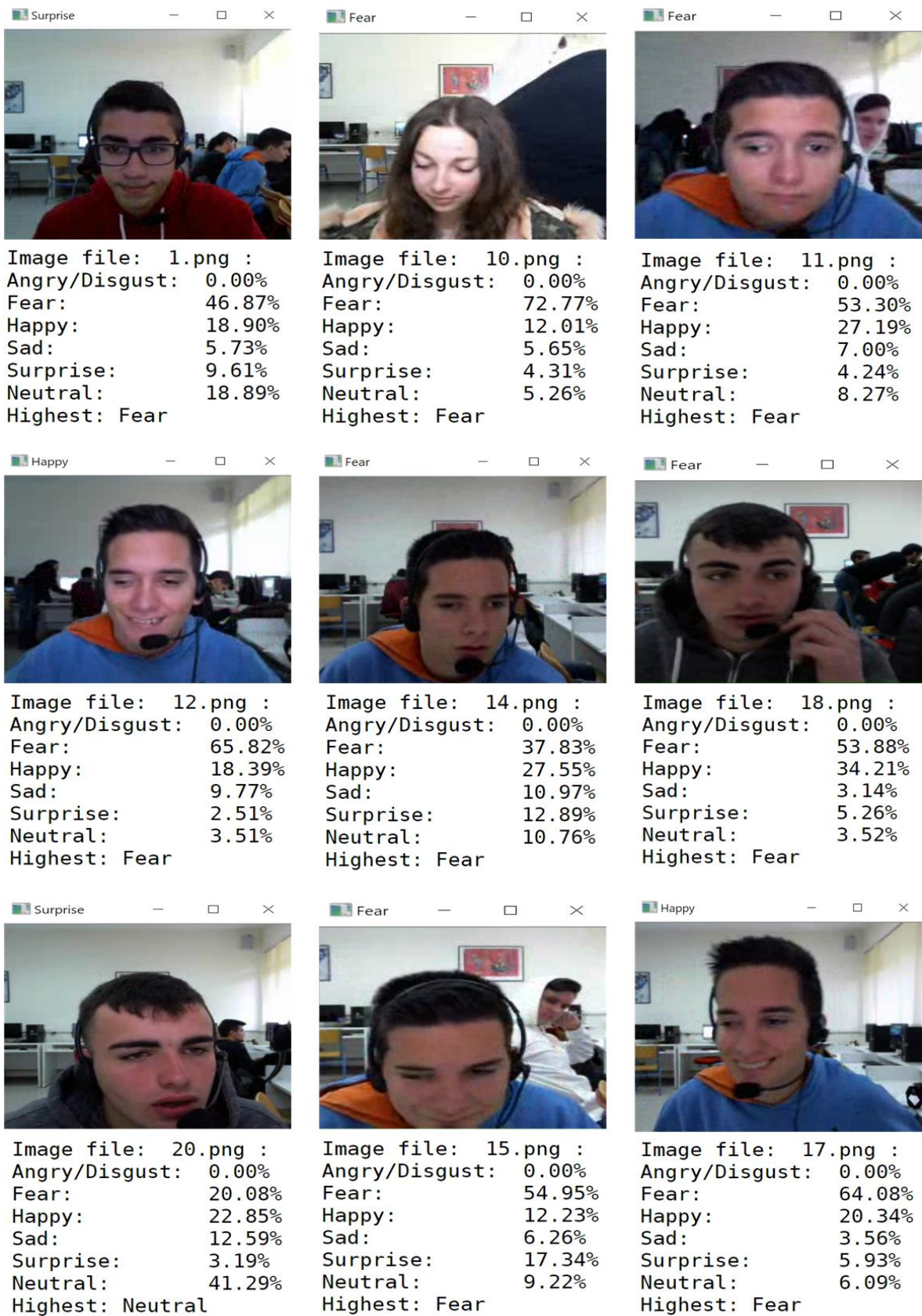


Figure 50: Output of the predicted unseen images for the fine-tuned Inception-v3 network

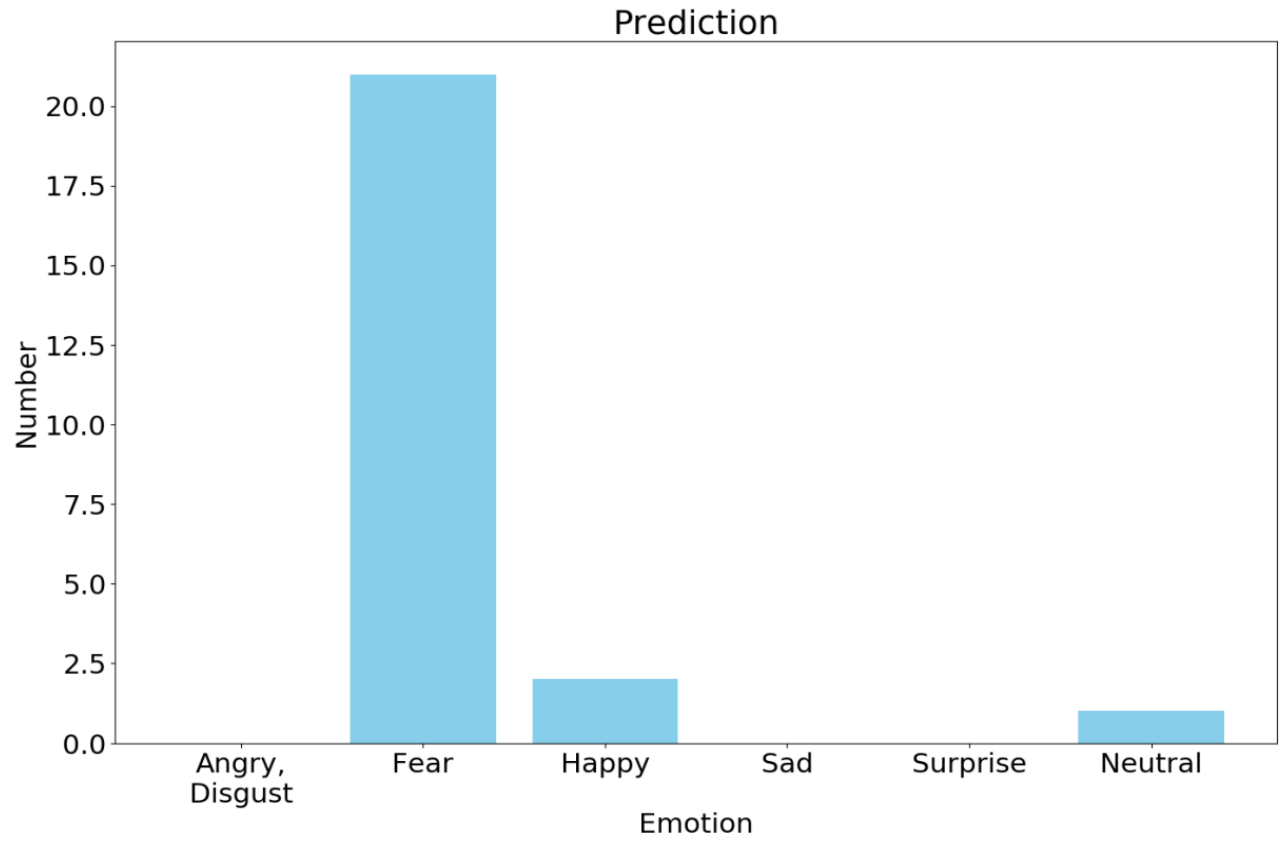


Figure 51: Prediction result of the unseen images for the fine-tuned Inception-v3 network

4.3 Comparison

In this section, the results of both networks will be compared to each other and discussed. The best-achieved performances of the networks will be compared lastly against the state-of-the-art solutions mentioned in section 2.3. Figure 52 compares the training performance of the fine-tuned networks. It can be seen that the VGG16 converged quicker to a global minimum than the Inception-v3 network and didn't underfit the data in any of the epochs. In addition, it can be seen that the validation loss of the VGG16 decreased quicker in contrast to the Inception-v3. As previously explored, the maximum validation accuracy achieved with the VGG16 was 71% and that of the inception v3 was 68%. This resulted in better accuracy of 3% for the VGG16 network. The learning rate of both of them decreased with the same smoothness as can be seen.

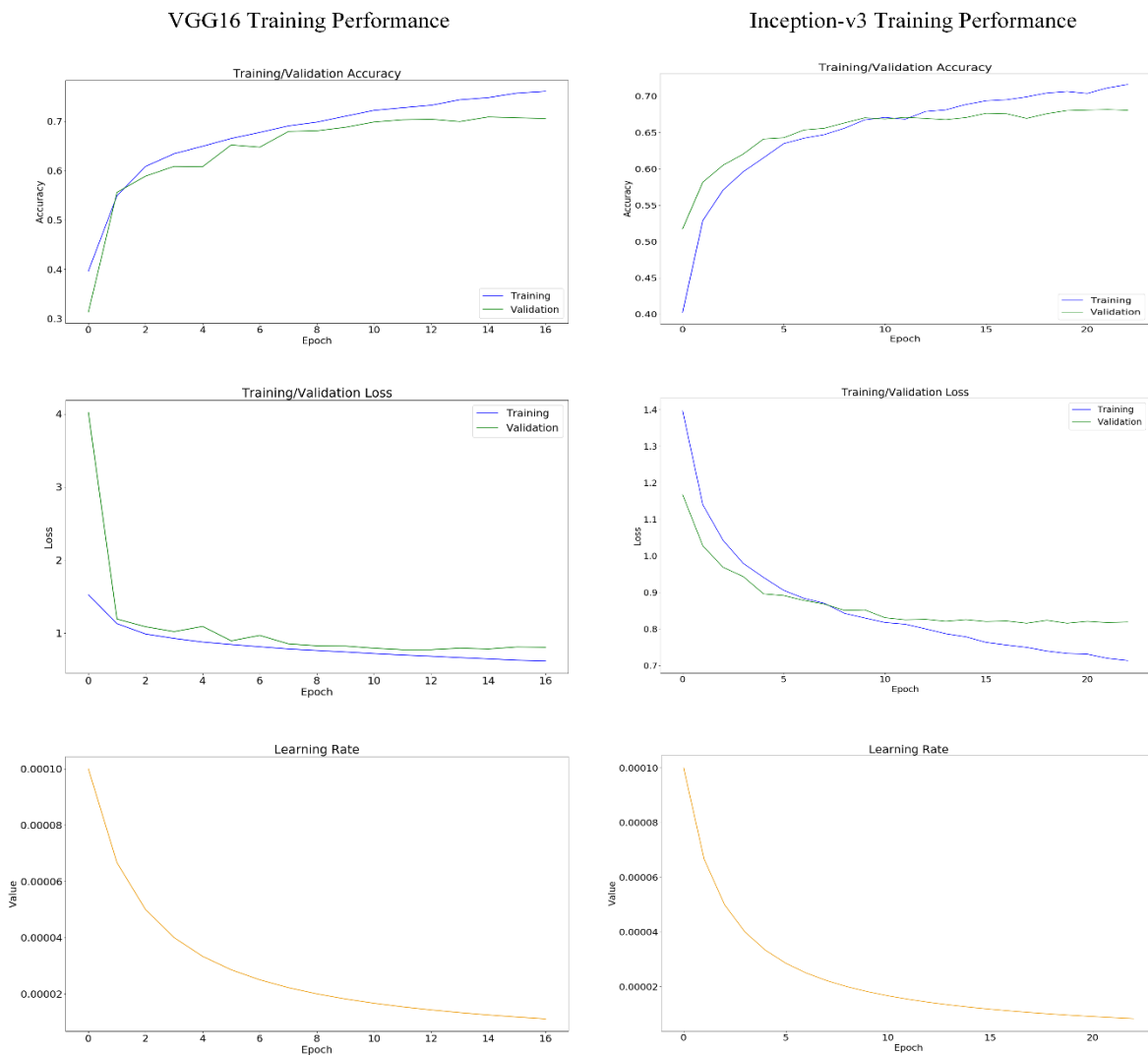
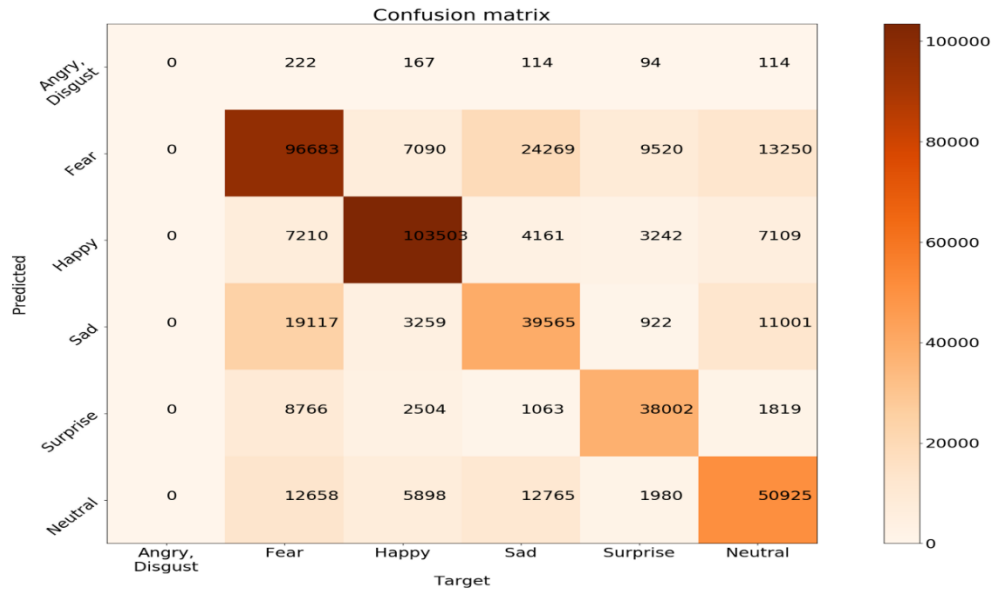


Figure 52: Comparison of the training performance

The evaluation of the validation process is shown in the confusion matrices depicted in Figure 53. The matrices show that both networks performed similarly while validating the network. This can be seen because both diagonals are similar to each other. The main difference is only the number in the cells, which is apparent because the VGG16 network converged earlier than the Inception-v3 and so the validation data was much less. Nevertheless, the matrices show that both networks were learning right.

VGG16 Validation Performance



Inception-v3 Validation Performance

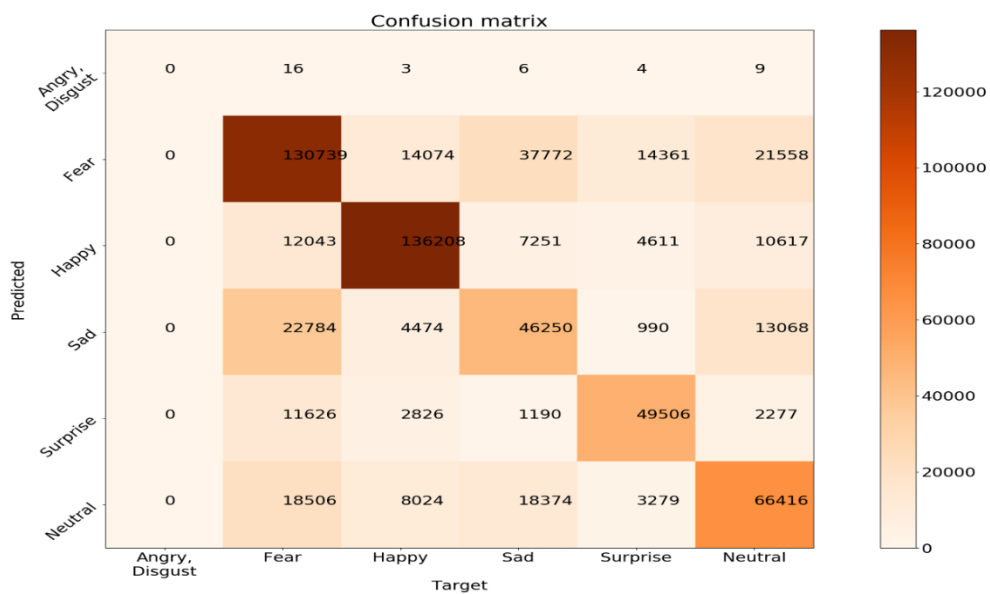


Figure 53: Comparison of the validation performance

Figure 54 compares the predictions of the networks. It can be seen that the Inception-v3 predicted some emotions more as Fear as the VGG16. Nevertheless, the network couldn't predict right any emotion as Happy, which was wrong because, as already mentioned, some pictures expressed these emotions. In contrast, the VGG16 could not recognize the emotion Sad, which was seen on some pictures. Both networks didn't recognize any emotion of Angry, Disgust, which was right because there were no such images expressing this emotion. Figure 55 compares the evaluation performance using the test set of the FER2013 dataset.

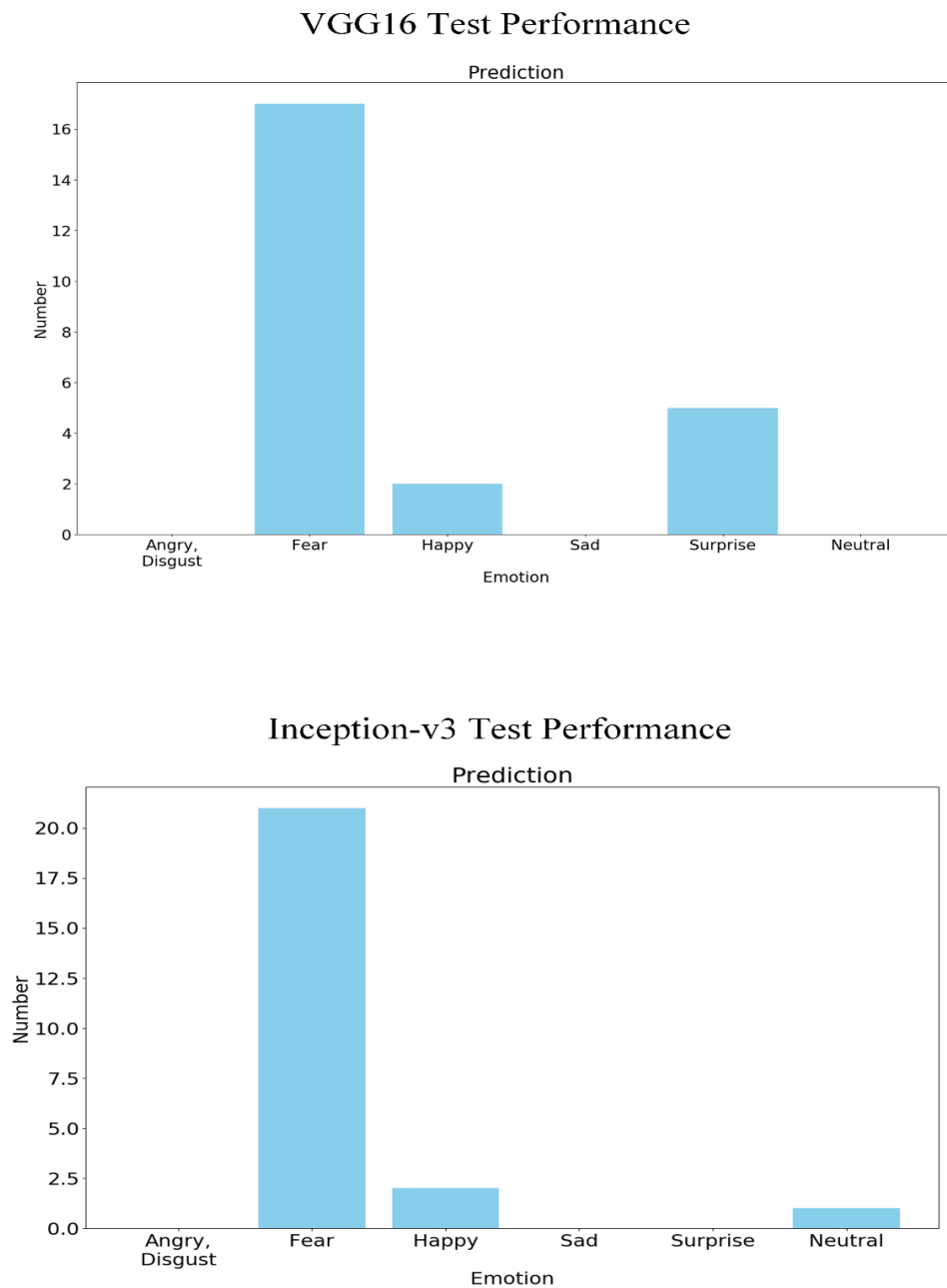


Figure 54: Comparison of the evaluation performance using the 24 unseen images

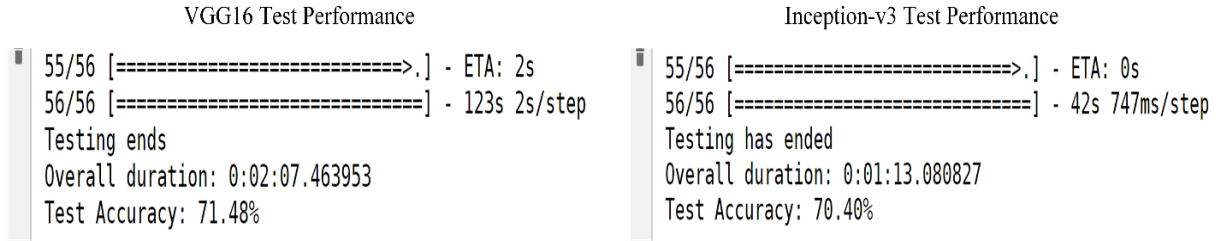


Figure 55: Comparison of the evaluation performance using the FER2013 test set

The above evaluation comparison using the test set of the FER2013 dataset depicts, that the performance achieved by the VGG16 network was 71,48% whereas the performance achieved by the Inception-v3 network was 70,40%. This concludes that the VGG16 network performs slightly better than the Inception-v3 network.

The execution time of the training and evaluation process (using the 24 unseen images) for the Inception-v3 network was higher than for the VGG16 network, which was to be expected since this network consisted of 23.907.110 parameters (Figure 31) in contrast to the VGG16 which consisted of only 14,719,814 parameters (Figure 29). In turn the execution time of the evaluation process while using the test set of the FER2013 dataset was lower for the VGG16 network. Based on the previous results it could be concluded that the VGG16 was more suitable for the FER task because of the following reasons:

- The VGG16 doesn't need much computational power as the Inception-v3
- The prediction quality of the VGG16 is slightly better than the Inception-v3
- The execution and prediction time (in most cases) are lower than the Inception-v3
- Underfitting cannot be observed while training the VGG16
- The loss decreases quicker in contrast to the Inception-v3
- Because of its less computation power, it is more suitable for embedded systems
- The training performance is better than for the Inception-v3

The last part compares the performance of both fine-tuned networks against the existing methods explained in section 2.3. This work has shown that using deeper network architectures consisting of Inception modules doesn't necessarily provide better performance. This can be concluded from the fact that 66.4% were achieved in the ILSVRC competition with the use of the FER2013 dataset, but more is achieved with the use of the adapted networks of the project. While trying different state-of-the-art solutions like the Inception-ResNet-v2, ResNet152, etc. it could be observed that the residual connections omit the overfitting problem,

but the performance was not quite good. The ensemble method was not considered in this work, but it is obvious that using an ensembled system of networks requires more computational power.

5. Conclusion

In this project, the goal was to show how a convolutional neural network is able to recognize facial emotions based on images of the FER2013 dataset. To compare the results, an older network and a state-of-the-art network were examined and their performance measured. To evaluate the prediction quality of the networks, 24 unseen images which got not edited before and the FER2013 test set were used. Each image of the 24 unseen images contained a student who was writing an exam and his or her mood was accordingly. Most of the emotions on the images could be perceived as Fear, Sad, Neutral and Happy based on the human intuition. The results of section 4 had shown that it does not mean when a pre-trained network performs well on one task like object detection or object recognition it performs the same or even better on another task like FER by fine-tuning the network architecture. The same applies to a network which comprises new technologies like inception blocks, this doesn't necessarily mean that this network performs better than an older one on the same dataset. The results have shown that regularization doesn't provide any time improvement to the performance. Nevertheless, this technique in most cases reduces the overfitting problem. Another method to omit overfitting is to use a bigger dataset either by augmenting an existing one or by using a different. In addition, the project has made it clear that a deeper network does not necessarily provide better performance than a shallow one. The conclusion of the work is that shallower CNN's achieve the same or even better results than a state-of-the-art network on the FER task and the execution time either for the training processes or the testing process is lower than the newer ones. This makes them more suitable for the implementation in embedded systems like the Raspberry and so on, which are electronic devices provide less computational power and resources. Nevertheless, for the future, the usage of bigger FER datasets containing better quality of images like the CMU MultiPIE (as mentioned earlier it is not free, but contains more than 750.000 images) could be tried out. Another approach could be to construct a shallow network from scratch, which will be able to provide better performances than it is possible with the pre-trained ones. In addition, other training techniques could be applied like ensembling multiple networks of the same or different type and combining the results to achieve better performance. Based on the result of the project it is more recommended to build a CNN from scratch consisting of a shallow architecture to achieve the best possible performance. If very deep networks get used, the required computational power and resources will increase significantly. In general, it is more recommended to get sufficient training data and to build a shallow network from scratch, instead of using a pre-trained and fine-tuned model. Despite there are plenty of techniques available to solve FER tasks, this keeps still a very important research topic. One research goal is to create deeper networks and reduce the number of computations and the required resources. It's interesting to see what the future holds to do such tasks better.

References

- [1] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).
- [2] SIMONYAN, Karen; ZISSERMAN, Andrew. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [3] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818-2826).
- [4] Ng, H. W., Nguyen, V. D., Vonikakis, V., & Winkler, S. (2015, November). Deep learning for emotion recognition on small datasets using transfer learning. In *Proceedings of the 2015 ACM on international conference on multimodal interaction* (pp. 443-449). ACM.
- [5] Lucey, P., Cohn, J. F., Kanade, T., Saragih, J., Ambadar, Z., & Matthews, I. (2010, June). The extended cohn-kanade dataset (ck+): A complete dataset for action unit and emotion-specified expression. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition-Workshops* (pp. 94-101). IEEE.
- [6] Gross, R., Matthews, I., Cohn, J., Kanade, T., & Baker, S. (2010). Multi-pie. *Image and Vision Computing*, 28(5), 807-813.
- [7] Li, S., & Deng, W. (2018). Deep facial expression recognition: A survey. *arXiv preprint arXiv:1804.08348*.
- [8] Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. *CVPR (1)*, 1, 511-518.
- [9] KAZEMI, Vahid; SULLIVAN, Josephine. One millisecond face alignment with an ensemble of regression trees. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014. p. 1867-1874.
- [10] Keller, James M., Michael R. Gray, and James A. Givens. "A fuzzy k-nearest neighbor algorithm." *IEEE transactions on systems, man, and cybernetics* 4 (1985): 580-585.
- [11] BOSER, Bernhard E.; GUYON, Isabelle M.; VAPNIK, Vladimir N. A training algorithm for optimal margin classifiers. In: *Proceedings of the fifth annual workshop on Computational learning theory*. ACM, 1992. p. 144-152.
- [12] Rish, I. (2001, August). An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence* (Vol. 3, No. 22, pp. 41-46).
- [13] GRAVES, Alex; MOHAMED, Abdel-rahman; HINTON, Geoffrey. Speech recognition with deep recurrent neural networks. In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013. p. 6645-6649.

- [14] TSAI, Chih-Fong; MCGARRY, Ken; TAIT, John. Image classification using hybrid neural networks. In: *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*. ACM, 2003. p. 431-432.
- [15] BOTTOU, Léon. Large-scale machine learning with stochastic gradient descent. In: *Proceedings of COMPSTAT'2010*. Physica-Verlag HD, 2010. p. 177-186.
- [16] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.
- [17] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- [18] MOLLAHOSSEINI, Ali; CHAN, David; MAHOOR, Mohammad H. Going deeper in facial expression recognition using deep neural networks. In: *2016 IEEE Winter conference on applications of computer vision (WACV)*. IEEE, 2016. p. 1-10.
- [19] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).
- [20] Jung, H., Lee, S., Yim, J., Park, S., & Kim, J. (2015). Joint fine-tuning in deep neural networks for facial expression recognition. In *Proceedings of the IEEE international conference on computer vision* (pp. 2983-2991).
- [21] Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., ... & Bengio, Y. (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*.
- [22] Seide, F., & Agarwal, A. (2016, August). CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 2135-2135). ACM.
- [23] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (pp. 265-283).
- [24] Mavadati, S. M., Mahoor, M. H., Bartlett, K., Trinh, P., & Cohn, J. F. (2013). Disfa: A spontaneous facial action intensity database. *IEEE Transactions on Affective Computing*, 4(2), 151-160.
- [25] Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009, June). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (pp. 248-255). Ieee.
- [26] Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017, February). Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.
- [27] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929-1958.

[28] IOFFE, Sergey; SZEGEDY, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Appendix

Python Code of the FER2013 Dataset Creation

```
#
#   Author:           Michail Tamvakeras
#   Postgraduate course: MSc in Intelligent Information Systems (MIIS)
#   University:       University of the Aegean
#   Department:       Information and Communication Systems Engineering
#
#   MSc Thesis:       Facial emotion recognition using deep learning
#

import numpy as np
import pandas as pd
import os
import datetime
import cv2
import h5py

class FER2013DatasetCreation():

    _dataset_path = './datasets/' # The path of the FER2013 dataset
    _num_emotions = 6 # The number of the emotion classes to use
    _image_size = 48 # The original FER2013 image properties (48x48
    grayscale images)

    def __init__(self, csv_file=None):

        assert csv_file is not None

        self._train_X = []
        self._train_Y = []
        self._valid_X = []
        self._valid_Y = []
        self._test_X = []
        self._test_Y = []

        # Checks if the log folder and subfolders exists
        if os.path.exists(self._dataset_path) == False:
            os.makedirs(self._dataset_path)

        self.import_csv(csv_file) # Reads the raw csv file content
        self.create_dataset() # Creates the dataset
        return

    def import_csv(self, file):
        """
        This method reads the FER2013 dataset from the provided path and creates the test,
        validation and training set.
        The FER2013 dataset is one of the most used dataset in Kaggle facial expression
        recognitions competitions:
        (https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data),
        and is divided into 60% training, 20% validation (PrivateTest) and 20% test
        (PublicTest) examples.
        The emotion classes are: 0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise,
        6=Neutral
        INPUT:
        """
```

```

- file: A file containing the dataset in csv format
"""

# Reads the csv file and extracts the emotion and pixels column
dataframe = pd.read_csv(file, delimiter=',', dtype='a')
classes = np.array(dataframe['emotion'], np.float)

# Creates the image matrices
data = np.array(dataframe['pixels'])
images = np.array([np.fromstring(image, np.uint8, sep=' ') for image in data])
del data
num_shape = int(np.sqrt(images.shape[-1]))
images.shape = (images.shape[0], num_shape, num_shape)

# Saves the train/test text
usage = np.array(dataframe['Usage'])

# Creates the training set
train_index = np.where(usage == 'PublicTest')
train_index = train_index[0][0]
self._train_X = images[:train_index]
self._train_Y = classes[:train_index]
self._train_Y = self._train_Y.reshape((self._train_Y.size, 1))

# Creates the validation set
valid_index = np.where(usage == 'PrivateTest')
valid_index = valid_index[0][0]
self._valid_X = images[train_index:valid_index]
self._valid_Y = classes[train_index:valid_index]
self._valid_Y = self._valid_Y.reshape((self._valid_Y.size, 1))

# Creates the test set
self._test_X = images[valid_index:]
self._test_Y = classes[valid_index:]
self._test_Y = self._test_Y.reshape((self._test_Y.size, 1))

# Displays the train/test set amount and the respective shapes
print("The FER2013 dataset consists of:")
print("-----")
print("[Training]\t\tImages:\t" + str(len(self._train_X)) + "\t" +
str(self._train_X.shape)
      + "\t\tLabels:\t" + str(self._train_Y.size) + "\t" + str(self._train_Y.shape)
      + "\n[Validation]\tImages:\t" + str(len(self._valid_X)) + "\t" +
str(self._valid_X.shape)
      + "\t\tLabels:\t" + str(self._valid_Y.size) + "\t" + str(self._valid_Y.shape)
      + "\n[Test]\t\t\tImages:\t" + str(len(self._test_X)) + "\t" +
str(self._test_X.shape)
      + "\t\tLabels:\t" + str(self._test_Y.size) + "\t" + str(self._test_Y.shape))
print("[Emotions]\t\t0:Angry, 1:Disgust, 2:Fear, 3:Happy, 4:Sad, 5:Surprise,
6:Neutral\n")
return

def create_dataset(self):
    """
    This method creates the fer2013 dataset, the files will be stored as HDF5 file.
    IMPORTANT: The labels angry and fear will be merged into one class, because their
    emotions are similar and each
    of them contains less than 1000 images: So the followigng classes will be created:
    0=Angry/Disgust, 1=Fear, 2=Happy, 3=Sad, 4=Surprise, 5=Neutral
    """

    # Creates the HDF5 file
    hdf5_path = "./hdf5/fer2013_dataset.hdf5"
    with h5py.File(hdf5_path, mode='w') as hdf5_file:

```

```

print("FER2013 dataset creation in progress...")
start = datetime.datetime.now().replace(microsecond=0)

# Creates the FER2013 training, validation and testing sub-datasets
# [TRAINING]
hdf5_file.create_dataset("training_img", (len(self._train_X), self._image_size,
self._image_size, 1), dtype='uint8', chunks=True)
hdf5_file.create_dataset("training_label", (len(self._train_Y), 1),
dtype='uint8', chunks=True)

# [VALIDATION]
hdf5_file.create_dataset("validation_img", (len(self._valid_X),
self._image_size, self._image_size, 1), dtype='uint8', chunks=True)
hdf5_file.create_dataset("validation_label", (len(self._valid_Y), 1),
dtype='uint8', chunks=True)

# [TESTING]
hdf5_file.create_dataset("testing_img", (len(self._test_X), self._image_size,
self._image_size, 1), dtype='uint8', chunks=True)
hdf5_file.create_dataset("testing_label", (len(self._test_Y), 1),
dtype='uint8', chunks=True)

# Creates each sub-dataset
self.dataset_creation(hdf5_file, 'training')
self.dataset_creation(hdf5_file, 'validation')
self.dataset_creation(hdf5_file, 'testing')
end = datetime.datetime.now().replace(microsecond=0)

print("FER2013 dataset creation completed.")
print("Duration: " + str((end - start)) + "\n")

# Displays the new labels
print("[NEW LABELS]\t\t0:Angry/Disgust, 1:Fear, 2:Happy, 3:Sad, 4:Surprise,
5:Neutral")
return

```

```

def dataset_creation(self, hdf5_file=None, operation='training'):
    """
    This method creates the dataset of the related operation e.g. training
    INPUT:
    - hdf5_file: The hdf5 file to store the data
    - operation: The dataset creation operation to execute (training, validation or
testing)
    RETURN:
    - max_examples: The number of the examples created
    """
    assert hdf5_file is not None

    print("Creating the "+operation+" set...")
    index = 0
    img_tensor = None
    max_examples = 0

    # Checks which data from the FER2013 to use
    op_X = -1
    op_Y = -1
    if operation is 'training':
        op_X = self._train_X
        op_Y = self._train_Y
    elif operation is 'validation':
        op_X = self._valid_X
        op_Y = self._valid_Y
    elif operation is 'testing':
        op_X = self._test_X

```

```

        op_Y = self._test_Y
    else:
        print("'INVALID OPERATION: Please provide which dataset operation to execute
i.e. training, validation or testing")
        return -1

    for i, (img, lab) in enumerate(zip(op_X, op_Y)):

        # Merges the emotions anger and disgust into one class, reduces the emotion
classes by one to keep
        # consistent with 0=Angry/Disgust, 1=Fear, 2=Happy, 3=Sad, 4=Surprise,
5=Neutral
        if (lab == 0 or lab == 1):
            lab = 1
        elif (lab > 0 and lab <= 6):
            lab -= 1

        # Stores the data (in tensor format) into the hdf5 dataset
img_tensor = self.image_to_tensor(img)
hdf5_file[operation+'_img'][index, :, :, :] = img_tensor
hdf5_file[operation+'_label'][index, :] = lab
index += 1
max_examples += 1
del (img); del (lab) # Deletes the temporary image and label lists to free the
memory

    return max_examples

def image_to_tensor(self, image):
    """
    This method transforms the image to a tensor
    INPUT:
    - image: The image to transform
    RETURN:
    - tens_image: The image tensor
    """
    tens_image = cv2.resize(image, dsize=(self._image_size, self._image_size))
    tens_image = tens_image.reshape(self._image_size, self._image_size, 1)
    return tens_image

if __name__ == '__main__':
    FER2013DatasetCreation('./datasets/fer2013.csv')

```

Python Code of the Custom Visualization Tools

```
#
# Author: Michail Tamvakeras
# Postgraduate course: MSc in Intelligent Information Systems (MIIS)
# University: University of the Aegean
# Department: Information and Communication Systems Engineering
#
# MSc Thesis: Facial emotion recognition using deep learning
#
```

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import numpy as np
```

```
def setup_plot(font_size=18, show_grid=True):
    """
    Initializes the plot
    INPUT:
    - font_size: The font size
    - show_grid: Flag to show/hide the grid
    """
    plt.rcParams.update({'font.size': font_size}) # Sets the font size
    plt.clf()
    if show_grid:
        plt.grid() # Shows the grid if desired
    return
```

```
def plot_function(interval, step, func=None, color='blue'):
    """
    This method plots the given function
    INPUT:
    - interval: The interval of the x-axis
    - step: The step for the x-axis
    - func: The function to plot
    - color: Color to print the function
    IMPORTANT: The function to plot, has to be defined before
    """
    assert func is not None

    x = np.arange(interval[0], interval[1], step)
    plt.plot(x, func(x), color=color)
    # plt.xlabel('x')
    # plt.ylabel('f(x)')
    plt.show()
    return
```

```
def plot_learning_rate_history(learning_rates=None):
    """
    This method plots the learning rate history
    INPUT:
    - training_losses: The learning rate history to plot
    """
    x_axis = [val for val in range(len(learning_rates))]
    plt.locator_params(integer=True) # Prints the x-axis values as integers
    plt.plot(x_axis, learning_rates, '#EE9A00') # Orange
    plt.title('Learning Rate')
    plt.xlabel('Epoch')
    plt.ylabel('Value')
    plt.show()
    return
```

```

def plot_loss_history(training_losses=None, validation_losses=None):
    """
    This method plots the loss history of the desired operations (training, validation,
    testing)
    INPUT:
    - training_losses: The given training losses to plot as a list
    - validation_losses: The given validation losses to plot as a list
    """
    legend = []
    x_axis = [val for val in range(len(training_losses))]
    plt.locator_params(integer=True) # Prints the x-axis values as integers
    if training_losses is not None: # Checks for plotting the training loss
        plt.plot(x_axis, training_losses, 'b-')
        legend.append('Training')
    if validation_losses is not None: # Checks for plotting the validation loss
        plt.plot(x_axis, validation_losses, 'g-')
        legend.append('Validation')
    plt.legend(legend, loc='upper right')
    plt.title('Training/Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.show()
    return

def plot_accuracy_history(training_acc=None, validation_acc=None):
    """
    This method plots the loss history of the desired operations (training, validation,
    testing)
    INPUT:
    - training_acc: The given training accuracy to plot as a list
    - validation_acc: The given validation accuracy to plot as a list
    """
    legend = []
    x_axis = [val for val in range(len(training_acc))]
    plt.locator_params(integer=True) # Prints the x-axis values as integers
    if training_acc is not None: # Checks for plotting the training loss
        plt.plot(x_axis, training_acc, 'b-')
        legend.append('Training')
    if validation_acc is not None: # Checks for plotting the validation loss
        plt.plot(x_axis, validation_acc, 'g-')
        legend.append('Validation')
    plt.legend(legend, loc='lower right')
    plt.title('Training/Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.show()
    return

def plot_barchart(x_values, y_values):
    """
    This method plots a bar chart of the given distribution
    INPUT:
    -
    """
    y = np.arange(len(x_values))
    plt.bar(y, y_values, align='center', color='SkyBlue')
    plt.xticks(y, x_values)
    plt.ylabel('Number')
    plt.xlabel('Emotion')
    plt.title('Prediction')
    plt.show()
    return

```

```

def plot_confusion_matrix(target_labels, predicted_labels, class_names, plot_names,
invalid_names,
    title='Confusion matrix', cmap=plt.cm.Oranges):
    """
    This function prints and plots the confusion matrix, based on the sklearn source:
    https://scikit-
    learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
    #sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py.
    INPUT:
    - target_labels: The target labels
    - predicted_labels: The predicted labels
    - class_names: The names of the emotion classes
    - plot_names: Names to plot
    - invalid_names: The labels which are invalid (labels that are not predicted)
    - title: Title of the confusion matrix
    - cmap: The color distribution of the confusion matrix
    """

    # Calculates the confusion matrix including invalid labels
    invalid_values = [-1] * len(invalid_names)
    cm = confusion_matrix(target_labels+invalid_names, predicted_labels+invalid_values,
labels=class_names)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(plot_names))
    plt.xticks(tick_marks, plot_names)
    plt.yticks(tick_marks, plot_names, rotation=45)

    # print(cm)
    # Prints row by row the values in the confusion matrix
    for i, name in zip(range(len(class_names)), class_names):
        for j in range(len(class_names)):
            plt.text(i,j, str(cm[i][j]))
    plt.tight_layout()
    plt.ylabel('Predicted')
    plt.xlabel('Target')
    plt.show()
    return

```

Python Code of the Main Program

```
#
#   Author:           Michail Tamvakeras
#   Postgraduate course: MSc in Intelligent Information Systems (MIIS)
#   University:       University of the Aegean
#   Department:       Information and Communication Systems Engineering
#
#   MSc Thesis:       Facial emotion recognition using deep learning
#

from FERManager import *
from FerVGG16 import *
from FerInceptionV3 import *

if __name__ == '__main__':
    # Creates the pre-trained Inception-v3 network
    inception_v3 = FerInceptionV3(
        64,           # Mini-batch size
        1E-4,        # Initial learning rate
        100,         # Max epochs
        'inception-v3', # The model name
        100,         # Image input size (orig: 299x299x3) IMPORTANT: For testing image
                    # size must be the original
        'rgb')

    # Creates the pre-trained VGG16 network
    vgg16 = FerVGG16(
        64,           # Mini-batch size
        1E-4,        # Initial learning rate
        100,         # Max epochs
        'vgg16',     # The model name
        100,         # Image input size (orig: 224x224x3) IMPORTANT: For testing image
                    # size must be the original
        'rgb')

    # Creates the FER neural network manager, who manages the networks and the whole FER
    # process
    nn_manager = FerManager()
    nn_manager.add_network(vgg16)
    nn_manager.add_network(inception_v3)

    # Executes the training process
    # nn_manager.start_training('vgg16')
    # nn_manager.start_training('inception-v3')

    # Executes the testing process
    nn_manager.test("vgg16")
    # nn_manager.test("inception-v3")

    # Predicts the emotion of the images of the specified folder
    # nn_manager.predict("./datasets/photos/", "vgg16")
    # nn_manager.predict("./datasets/photos/", "inception-v3")
```

Python Code of the Facial Emotion Recognition Manager (FerManager)

```
#
# Author: Michail Tamvakeras
# Postgraduate course: MSc in Intelligent Information Systems (MIIS)
# University: University of the Aegean
# Department: Information and Communication Systems Engineering
#
# MSc Thesis: Facial emotion recognition using deep learning
#

import os
import cv2
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # Just disables the warning, doesn't enable
AVX/FMA

from FerVGG16 import *
from FerInceptionV3 import *
import tensorflow as tf
import keras.backend.tensorflow_backend as tf_backend
from VisualizationTools import plot_barchart, setup_plot

class FerManager:
    """
    This class contains the management of the facial emotion recognition process (FER).
    """

    _num_emotions = 6 # Emotions (1=Angry/Disgust, 2=Fear, 3=Happy, 4=Sad,
5=Surprise, 6=Neutral)
    _log_dir = './logs/'

    def __init__(self):
        self._nn_dict = {} # Creates an empty network dictionary
        return

    def enable_gpu_support(self):
        # Checks for GPU support on the system
        self._gpus = tf_backend._get_available_gpus()
        if len(self._gpus) is not 0:
            tf.Session(config=tf.ConfigProto(log_device_placement=True)) # logs the GPU
            gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.8,
allow_growth=True)
            sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))
            tf_backend.set_session(sess)
        else:
            print("No GPU found on system, execution will be done using the CPU only!!!")
        return

    def add_network(self, network=None):
        """
        This method adds a new network to the network list
        network: The network tha has to be added to the network list
        INPUT:
        - network: The network
        """

        # Checks if the given net is subclass of CNNetwork
        assert isinstance(type(network), CNNetwork)
        self._nn_dict[network.get_name()] = network
        return
```

```

def get_network_by_name(self, model_name=None):
    """
    This method loads the stored network by its name from the neural network list
    model_name: The name of the neural network to load
    INPUT:
    - model_name: The name of the network to load
    RETURN:
    - The specified network in the network list
    """
    assert model_name is not None
    return self._nn_dict[model_name]

def code_to_emotion(self, code):
    """
    This method converts the emotion number to the appropriate emotion name
    INPUT:
    - code: The integer code of the emotion
    RETURN:
    - The string name of the respective emotion code, otherwise None
    """
    emotion_names = {0: "Angry/Disgust",
                     1: "Fear",
                     2: "Happy",
                     3: "Sad",
                     4: "Surprise",
                     5: "Neutral"}

    if code >=0 and code <=5:
        return emotion_names.get(code)
    return None

def start_training(self, network_type):
    """
    This method executes the training and validation process of the models
    INPUT:
    - network_type: The type of the network to train
    """

    self.enable_gpu_support() # Enables the GPU support when it's available

    if network_type not in self._nn_dict:
        raise ValueError("The given network could not be found in the neural network
list!!!")
    network = self.get_network_by_name(network_type)

    # Checks if the log folder for the VGG16 network exists
    if network.get_name() == 'vgg16':
        if os.path.exists(self._log_dir + "vgg16") == False:
            os.makedirs(self._log_dir + "/vgg16/train/")

    # Checks if the log folder for the Inception-v3 network exists
    if network.get_name() == 'inception-v3':
        if os.path.exists(self._log_dir + "inception_v3") == False:
            os.makedirs(self._log_dir + "/inception_v3/train/")

    print("Training of the "+network_type+" network starts...")
    network.build(self._num_emotions)
    network.training(augmentation=True, early_stopping=True, decay_rate=0.5) # Decays
50% the learning rate
    return

```

```

def predict(self, dataset_path, network=None):
    """
    This method classifies an unknown image
    INPUT:
    - dataset_path: The path of the dataset
    - network: The network to predict
    """
    assert network is not None

    image_list = os.listdir(dataset_path)
    print("Prediction using the", network, "model...")
    vgg16 = None
    test_model_dir = ""
    inception_resnet_v2 = None

    # Checks for the VGG16 network
    if network == 'vgg16':
        vgg16 = self.get_network_by_name(network)
        test_model_dir = "./logs/vgg16/train/"
    # Checks for the Inception-v3 network
    elif network == 'inception-v3':
        inception_resnet_v2 = self.get_network_by_name(network)
        test_model_dir = "./logs/inception_v3/train/"
    else:
        print("No specified network for testing found!!!")
        return

    # Checks if the test directory contains a test checkpoint
    if os.listdir(test_model_dir):
        file = [file for file in os.listdir(test_model_dir)]
        test_model = load_model(test_model_dir + str(file[0]))
    else:
        print("No saved models available, prediction cannot proceed!!!")
        return
    emotion_predictions = []

    # Predicts the unknown images
    for img_file in image_list:
        # Reads every image from the folder, converts it to grayscale
        img_orig = cv2.imread(dataset_path + img_file)
        img_gray = cv2.imread(dataset_path + img_file, 0)

        predictions = []; calc_predictions = []; y_label = []
        if vgg16:
            predictions = vgg16.predict(img_gray, test_model)
        elif inception_resnet_v2:
            predictions = inception_resnet_v2.predict(img_gray, test_model)
        # print(predictions)

        print("Image file: ", img_file, ":")
        for code, percent in enumerate(predictions):
            emotion = self.code_to_emotion(code) # Converts the code to the emotion

            # Checks for which emotion/percentage to print
            if code == 0: print(emotion + ":\t%1.2f%%" % (np.float(percent)))
            elif code == 1: print(emotion + ":\t\t\t%1.2f%%" % (np.float(percent)))
            elif code == 2: print(emotion + ":\t\t\t\t%1.2f%%" % (np.float(percent)))
            elif code == 3: print(emotion + ":\t\t\t\t\t%1.2f%%" % (np.float(percent)))
            elif code == 4: print(emotion + ":\t\t\t\t\t\t%1.2f%%" % (np.float(percent)))
            elif code == 5: print(emotion + ":\t\t\t\t\t\t\t%1.2f%%" % (np.float(percent)))
            else: print("Invalid Code!!!")
            calc_predictions.append(percent)

        highest = self.code_to_emotion(np.argmax(calc_predictions))
        print("Highest:", highest, "\n")

```

name

```

emotion_predictions.append(highest)

cv2.imshow(highest, img_orig)
cv2.waitKey(0)
cv2.destroyAllWindows()

self.plot_statistics(emotion_predictions) # Plots the statistics of the prediction
return

def test(self, network=None):
    """
    This method uses the test set to evaluate the given network
    INPUT:
    - dataset_path: The path of the dataset
    - network: The network to predict
    """
    assert network is not None

    self.enable_gpu_support() # Enables the GPU support if it's available

    print("Testing using the", network, "model...")
    vgg16 = None
    test_model_dir = ""
    inception_resnet_v2 = None

    # Checks for the VGG16 network
    if network == 'vgg16':
        vgg16 = self.get_network_by_name(network)
        test_model_dir = "./logs/vgg16/train/"
    # Checks for the Inception-v3 network
    elif network == 'inception-v3':
        inception_resnet_v2 = self.get_network_by_name(network)
        test_model_dir = "./logs/inception_v3/train/"
    else:
        print("No specified network for testing found!!!")
        return

    # Checks if the test directory contains a test checkpoint
    if os.listdir(test_model_dir):
        file = [file for file in os.listdir(test_model_dir)]
        test_model = load_model(test_model_dir + str(file[0]))
    else:
        print("No saved models available, prediction cannot proceed!!!")
        return

    result = []
    if vgg16:
        result = vgg16.testing(self._num_emotions, test_model)
    elif inception_resnet_v2:
        result = inception_resnet_v2.testing(self._num_emotions, test_model)
    print("Test Accuracy: {:.2f}%".format(result))
    return

def plot_statistics(self, results):
    """
    This method prints the statistics of the prediction
    INPUT:
    - results: The results of the prediction to plot
    """

    # Prints a bar chart
    emotion_names = ["Angry,\nDisgust", "Fear", "Happy", "Sad", "Surprise",
"Neutral"]
    occurrence = []

```

```
for emotion in emotion_names:
    number = results.count(emotion)
    occurrence.append(number)
setup_plot(font_size=22, show_grid=False)
plot_barchart(emotion_names, occurrence)
return
```

Python Code of the Base Class of the Networks (CNNNetwork)

```
#
#   Author:           Michail Tamvakeras
#   Postgraduate course: MSc in Intelligent Information Systems (MIIS)
#   University:       University of the Aegean
#   Department:       Information and Communication Systems Engineering
#
#   MSc Thesis:       Facial emotion recognition using deep learning
#

import cv2
import numpy as np
import h5py

from keras.callbacks import Callback
from keras import backend as K
import tensorflow as tf

class CNNNetwork:
    """
    This class is the base class of the CNN's, which defines the common properties and
    operations.
    """

    _logs = "./Logs/" # Checkpoint and log directories
    fer_img_dataset_file = "./hdf5/fer2013_dataset.hdf5" # Path to the datasets

    def __init__(self, mini_batch_size, learning_rate, max_epochs, net_name, image_size,
                 image_type):

        self._sess = None
        self._mini_batch_size = mini_batch_size # Mini-batch size
        self._initial_learning_rate = learning_rate # Initial learning rate
        self._max_epochs = max_epochs # Number of epochs
        self._net_name = net_name # Name of the network type
        self._image_size = image_size # Image size of the input
        self._image_type = image_type # Type of the image (RGB or Gray)
        return

    # ABSTRACT METHODS, THEY HAVE TO BE IMPLEMENTED IN THE SUBCLASSES
    def build(self, *args, **kwargs):
        raise NotImplementedError("Subclass has to specify build method!!!")

    def training(self):
        raise NotImplementedError("Subclass has to specify the training method!!!")

    def predict(self, data):
        raise NotImplementedError("Subclass has to specify the prediction method!!!")

    def testing(self):
        raise NotImplementedError("Subclass has to specify the test method!!!")

    def get_name(self):
        return self._net_name

    def get_max_dataset_values(self):
        """
```

```

This method returns the amount of each dataset (training, validation and test)
RETURN:
- max_train: Maximum number of training examples
- max_valid: Maximum number of validation examples
- max_test: Maximum number of test examples
"""
dataset_file = self.fer_img_dataset_file
with h5py.File(dataset_file, mode='r') as hdf5_file:
    max_train = hdf5_file['training_img'].shape[0]
    max_valid = hdf5_file['validation_img'].shape[0]
    max_test = hdf5_file['testing_img'].shape[0]
return max_train, max_valid, max_test

def get_successive_batch(self, current_index, mini_batch_type='training'):
    """
    This method loads the next mini-batch of the given mini-batch size and type in
    successive order.
    INPUT:
    - current_index: The current example index given
    - mini_batch_type: The type of the dataset (training, validation or testing) to
    access
    RETURN:
    - images: The mini-batch containing the images in successive order
    - labels: The mini-batch containing the labels in successive order
    - index: The index of the current example of the dataset
    """
    dataset_file = self.fer_img_dataset_file

    # Gets the next mini-batch of the image dataset
    with h5py.File(dataset_file, mode='r') as hdf5_file:

        # Loads the mini-batch for training
        dataset_img_type = mini_batch_type + "_img"
        dataset_label_type = mini_batch_type + "_label"
        max_examples = hdf5_file[dataset_img_type].shape[0]

        index = None
        images = None
        labels = None
        diff = max_examples - current_index

        # Last mini-batch
        if diff <= self._mini_batch_size:
            images = hdf5_file[dataset_img_type][current_index:max_examples]
            labels = hdf5_file[dataset_label_type][current_index:max_examples]
            index = -1
        # There are still mini-batches to proceed
        elif diff > self._mini_batch_size:
            images = hdf5_file[dataset_img_type][current_index:current_index +
self._mini_batch_size]
            labels = hdf5_file[dataset_label_type][current_index:current_index +
self._mini_batch_size]
            index = current_index + self._mini_batch_size
        return images, labels, index

def preprocess_images(self, img_batch=None, label_batch=None):
    """
    This method pre-process the mini-batch to get fit into the network
    INPUT:
    - img_batch: Image batch to process
    - label_batch: The labels to process
    RETURN:
    - preproc_image_batch: The pre-processed images for the network
    - preproc_label_batch: The pre-processed labels for the network

```

```

"""
assert img_batch is not None
assert label_batch is not None

preproc_label_batch = np.zeros((label_batch.shape[0], 1), dtype = np.uint8)
preproc_image_batch = None

# Checks if the given image is a grayscale or RGB image
# GRAY:
if self._image_type is 'gray':
    # Creates a list of the appropriate size and appends the resized images and the
labels
    preproc_image_batch = np.zeros((img_batch.shape[0], self._image_size,
self._image_size, 1), dtype=np.float32)
    for i, (img, lab) in enumerate(zip(img_batch, label_batch)):
        img = cv2.resize(img, dsize=(self._image_size, self._image_size)) #
Resizes the image
        img = np.float32(img/255) # Normalizes the image in the range [0, 1]
        img = np.reshape(img, (img.shape[0], img.shape[1], 1))
        preproc_image_batch[i, :, :, :] = img # Stores the image
        preproc_label_batch[i, :] = lab
# RGB:
elif self._image_type is 'rgb':
    # Creates a list of the appropriate size and appends the resized images
    preproc_image_batch = np.zeros((img_batch.shape[0], self._image_size,
self._image_size, 3), dtype=np.float32)
    for i, (img, lab) in enumerate(zip(img_batch, label_batch)):
        img = cv2.resize(img, dsize=(self._image_size, self._image_size))
        img = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
        img = np.float32(img/255)
        preproc_image_batch[i, :, :, :] = img
        preproc_label_batch[i, :] = lab
return preproc_image_batch, preproc_label_batch

def preprocess_image(self, image=None):
    """
    This method pre-process an image, to fit into the network.
    INPUT:
    - image: Image to process
    RETURN:
    - new_image: The pre-processed image for the network
    """
assert image is not None

    new_image = None
    proc_img = None
    filter = np.array([[ -1, -1, -1], [-1, 9, -1], [-1, -1, -1]]) # Filter to apply to
sharpen the images
    proc_img = cv2.resize(image, dsize=(self._image_size, self._image_size))
    proc_img = cv2.cvtColor(proc_img, cv2.COLOR_GRAY2RGB)
    proc_img = cv2.filter2D(proc_img, -1, filter) # Apply filter to sharpen the image
    proc_img = np.float32(proc_img/255)
    new_image = np.zeros((1, self._image_size, self._image_size, 3), dtype=np.float32)
    new_image[0,:,:,:] = proc_img
return new_image

# CUSTOM CALLBACK CLASS
class PredictionHistory(Callback):
    # Based on: https://stackoverflow.com/questions/47079111/create-keras-callback-to-save-model-predictions-and-targets-for-each-batch-durin

    def __init__(self):
        Callback.__init__(self)
        self._targets = []

```

```
self._predictions = []
self._true = tf.Variable(0., validate_shape=False)
self._pred = tf.Variable(0., validate_shape=False)

def on_batch_end(self, batch, logs=None):
    self._targets.append(np.argmax(K.eval(self._true), axis=1))
    self._predictions.append(np.argmax(K.eval(self._pred), axis=1))
```

Python Code of the Fine-Tuned VGG16 Network (FerVGG16)

```
#
# Author: Michail Tamvakeras
# Postgraduate course: MSc in Intelligent Information Systems (MIIS)
# University: University of the Aegean
# Department: Information and Communication Systems Engineering
#
# MSc Thesis: Facial emotion recognition using deep learning
#

import datetime
import numpy as np
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # Just disables the warning, doesn't enable
AVX/FMA

from CNNetwork import CNNetwork
from keras.models import Model, load_model
from keras.applications.vgg16 import VGG16
from keras.layers import Dense, Dropout, BatchNormalization
from keras.utils import to_categorical
from keras.optimizers import Adam
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ModelCheckpoint, EarlyStopping, LearningRateScheduler
import tensorflow as tf
from VisualizationTools import plot_loss_history, plot_accuracy_history,
plot_learning_rate_history, plot_confusion_matrix, setup_plot

class FerVGG16(CNNetwork):
    """
    This class is derived from the base class CNNetwork, which implements a custom
    training and testing behaviour of the VGG16 model.
    """
    _saved_model_dir = './logs/vgg16/'

    def __init__(self, mini_batch_size=10, initial_learning_rate=0.01, max_epochs=10,
                 net_name=None, image_size=48, image_type='rgb'):

        assert net_name is not None

        # Calls the base class first
        CNNetwork.__init__(self, mini_batch_size, initial_learning_rate, max_epochs,
                           net_name, image_size, image_type)
        return

    def build(self, num_emotions):
        """
        This method builds the adapted VGG16 model
        INPUT:
        - num_emotions: The amount of the emotion classes:
          0=Angry/Disgust, 1=Fear, 2=Happy, 3=Sad, 4=Surprise, 5=Neutral
        """
        assert num_emotions == 6
        self._num_emotions = num_emotions # Stores the number of the emotion classes

        # Setups the model
        vgg16 = VGG16(
            input_shape=[self._image_size, self._image_size, 3], # The image size the
            network requires
            weights='imagenet', # The pre-trained weights from the ImageNet
            pooling='max', # Uses a max-pooling layer as last pooling layer
```

```

        include_top=False # Excludes the last layer of the network
    )

    for layer in vgg16.layers[:3]: # Freezes all layers until the penultimate
convolutional layer
        layer.trainable = False
    for layer in vgg16.layers[3:]: # Unfreezes all layers until the penultimate
convolutional layer
        layer.trainable = True

    # Adapts the last layers to classify the six emotion classes
    batchnorm = BatchNormalization()(vgg16.output)
    dropout_layer = Dropout(0.2)(batchnorm) # Disables randomly 20% of the neurons
    predictions = Dense(units=num_emotions, activation='softmax',
name='predictions')(dropout_layer)

    self._learning_rate = self._initial_learning_rate # Stores the initial learning
rate

    # Configures the learning process
    model = Model(inputs=vgg16.input, outputs=predictions)
    model.compile(
to use
        optimizer=Adam(lr=self._learning_rate), # The optimizer and the learning rate
        loss='categorical_crossentropy', # The loss function
        metrics=['accuracy'] # The metrics to calculate
    )
    print(model.summary()) # Prints the model structure

    # Checks if there is a saved model available, if not the new one gets stored
    train_save_path = self._saved_model_dir + "train/"
    files = [file for file in os.listdir(train_save_path)]
    if len(files) is 0:
        model.save(train_save_path + "init.h5")
    return

def create_generator(self, batch_size, mini_batch_type='training', aug=None):
    """
    This method creates a generator to reads the mini-batch data from disk
    and returns them to be feed into the network
    INPUT:
    - batch_size: The batch size to of the images
    - mini_batch_type: Either training or validation
    - aug: If the data should be augmented or not. Has to be None for validation
    IMPORTANT: The data reading process has to be unlimited and only specified by the
epoch number
    """
    index = 0
    while True:
        # Reads the images and labels from the hdf5 file in successive order
        images, labels, index = self.get_successive_batch(index, mini_batch_type)

        # Preprocess the images if necessary
        if images[0].shape[1] != self._image_size or images[0].shape[2] !=
self._image_size:
            images, labels = self.preprocess_images(images, labels)

        one_hot_labels = to_categorical(labels, self._num_emotions)
        # If augmentation is desired
        if aug is not None:
            (images, one_hot_labels) = next(aug.flow(images, one_hot_labels,
batch_size=batch_size))
        if index is -1: # Resets the index to the start again and checks for learning
rate decay
            index = 0

```

```

        yield(images, one_hot_labels) # Returns the images and labels to the calling
method

# CUSTOM CALLBACK-METHOD
def rate_decay(self, epoch):
    """
    This callback method calculates the new learning rate to apply learning rate decay
    INPUT:
    - epoch: The current epoch
    """
    self._learning_rate = self._initial_learning_rate / (
        1 + self._decay_rate * epoch) # Decays the learning rate
    return np.float32(self._learning_rate)

def training(self, augmentation=True, early_stopping=True, decay_rate=-1):
    """
    This method trains the network
    INPUT:
    - augmentation: Flag which determines if the training set gets augmented or not
    - early_stopping: Flag to apply early stopping
    - decay_rate: The learning decay rate, if -1 not learning rate decay gets applied
    """
    print("Training starts...")

    max_train, max_valid, _ = self.get_max_dataset_values() # Gets the amount of
training examples
    train_save_path = self._saved_model_dir + "train/"
    start = datetime.datetime.now()

    # Checks if augmentation is desired
    if augmentation:
        # Creates an ImageDataGenerator which applies augmentation
        aug = ImageDataGenerator(rotation_range=25, zoom_range=0.17,
width_shift_range=0.2,
height_shift_range=0.2, shear_range=0.15, horizontal_flip=True,
fill_mode="nearest")
    else:
        aug = None

    # Creates the training and validation generator to read the data
    trainGen = self.create_generator(self._mini_batch_size, 'training', aug=aug)
    validGen = self.create_generator(self._mini_batch_size, 'validation', aug=None)

    # Loads a saved model
    files = [file for file in os.listdir(train_save_path)]
    model = load_model(train_save_path + files[0])

    # Setups the callback functions
    callback_list = []
    checkpoint = ModelCheckpoint(train_save_path+'epoch-{epoch:02d}-
val_acc_{val_acc:.2f}.hdf5', monitor='val_acc', verbose=1, save_best_only=True, mode='max')
    # Checks if learning rate decay is desired
    if decay_rate is not -1:
        self._decay_rate = decay_rate
        learning_rate_decay = LearningRateScheduler(self.rate_decay)
        callback_list.append(learning_rate_decay)
    # Checks if early stopping is desired
    if early_stopping:
        early_stop = EarlyStopping(monitor='val_loss', min_delta=0.001, patience=5,
restore_best_weights=True)
        callback_list.append(early_stop)
    callback_list.append(checkpoint)

```

```

# Creates a custom callback function to get the data for the confusion matrix
predictions_cbk = self.PredictionHistory()
fetches = [tf.assign(predictions_cbk._true, model.targets[0],
validate_shape=False),
tf.assign(predictions_cbk._pred, model.outputs[0],
validate_shape=False)]
model._function_kwargs = {'fetches': fetches}
callback_list.append(predictions_cbk)

# Starts training the network
history = model.fit_generator(
    trainGen,
    steps_per_epoch=max_train // self._mini_batch_size,
    validation_data=validGen,
    validation_steps=max_valid // self._mini_batch_size,
    epochs=self._max_epochs,
    callbacks=callback_list)

# PLOTS THE RESULTS
setup_plot(font_size=22, show_grid=False)
plot_loss_history(history.history['loss'], history.history['val_loss']) # Plots
the loss history
plot_accuracy_history(history.history['acc'], history.history['val_acc']) # Plots
the accuracy history
plot_learning_rate_history(history.history['lr']) # Plots the learning rate
history

# Plots the confusion matrix
class_labels = ["Angry/Disgust", "Fear", "Happy", "Sad", "Surprise", "Neutral"]
plot_names=["Angry,\nDisgust", "Fear", "Happy", "Sad", "Surprise", "Neutral"]
target_list = []; prediction_list = []
# Create the target and prediction lists for the confusion matrix
for item in predictions_cbk._targets:
    for val in item:
        target_list.append(self.code_to_emotion(val))
for item in predictions_cbk._predictions:
    for val in item:
        prediction_list.append(self.code_to_emotion(val))
invalid_labels = [n for n in class_labels if n not in target_list and n not in
prediction_list] # Reads the invalid values
plot_confusion_matrix(target_list, prediction_list, class_labels, plot_names,
invalid_labels)

# Prints the result after one epoch
end = datetime.datetime.now()
print("Training ends")
print("Overall duration: " + str(end - start))
return

def predict(self, image, model):
    """
    This method does the prediction of unknown images
    INPUT:
    - image: The image to predict
    - model: Model to test
    RETURN:
    - The accuracy of the classification step
    """
    if image.shape[0] != self._image_size or image.shape[1] != self._image_size:
        image = self.preprocess_image(image)

    # Executes the prediction
    predictions = model.predict(image)
    percentages = predictions * 100.0
    return percentages[0]

```

```

def testing(self, num_emotions, model):
    """
    This method tests the network
    INPUT:
    - num_emotions: The amount of the emotion classes:
      0=Angry/Disgust, 1=Fear, 2=Happy, 3=Sad, 4=Surprise, 5=Neutral
    """
    assert num_emotions == 6
    self._num_emotions = num_emotions # Stores the number of the emotion classes

    print("Testing starts...")

    _, _, max_test = self.get_max_dataset_values() # Gets the amount of testing
examples
    train_save_path = self._saved_model_dir + "train/"
    start = datetime.datetime.now()

    # Creates the testing generator to read the data
    testGen = self.create_generator(self._mini_batch_size, 'testing')

    # Loads a saved model
    files = [file for file in os.listdir(train_save_path)]
    model = load_model(train_save_path + files[0])

    # Executes the testing process
    predictions = model.evaluate_generator(
        testGen,
        steps=max_test // self._mini_batch_size,
        verbose=1,
        workers=1)

    end = datetime.datetime.now()
    print("Testing has ended")
    print("Overall duration: " + str(end - start))
    return predictions[1] * 100.0

def code_to_emotion(self, code):
    """
    This method converts the emotion number to the appropriate emotion name
    INPUT:
    - code: The integer code of the emotion
    RETURN:
    - The string name of the respective emotion code, otherwise None
    """
    emotion_names = {0: "Angry/Disgust",
                     1: "Fear",
                     2: "Happy",
                     3: "Sad",
                     4: "Surprise",
                     5: "Neutral"}

    if code >=0 and code <=5:
        return emotion_names.get(code)
    return None

```

Python Code of the Fine-Tuned Inception-v3 Network (FerInceptionV3)

```
#
# Author: Michail Tamvakeras
# Postgraduate course: MSc in Intelligent Information Systems (MIIS)
# University: University of the Aegean
# Department: Information and Communication Systems Engineering
#
# MSc Thesis: Facial emotion recognition using deep learning
#

import datetime
import numpy as np
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # Just disables the warning, doesn't enable
AVX/FMA

from CNNetwork import CNNetwork
from keras.models import Model, load_model
from keras.applications.inception_v3 import InceptionV3
from keras.layers import Dense
from keras.utils import to_categorical
from keras.optimizers import Adam
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ModelCheckpoint, EarlyStopping, LearningRateScheduler
import tensorflow as tf
from VisualizationTools import plot_loss_history, plot_accuracy_history,
plot_learning_rate_history, plot_confusion_matrix, setup_plot

class FerInceptionV3(CNNetwork):
    """
    This class is derived from the base class CNNetwork, which implements a custom
    training and evaluation behaviour of the Inception-v3 model.
    """
    _saved_model_dir = './logs/inception_v3/'

    def __init__(self, mini_batch_size=10, initial_learning_rate=0.01, max_epochs=10,
                 net_name=None, image_size=48, image_type='rgb'):

        assert net_name is not None

        # Calls the base class first
        CNNetwork.__init__(self, mini_batch_size, initial_learning_rate, max_epochs,
                           net_name, image_size, image_type)
        return

    def build(self, num_emotions):
        """
        This method builds the adapted Inception-v3 model
        INPUT:
        - num_emotions: The amount of the emotion classes: 0=Angry/Disgust, 1=Fear,
        2=Happy, 3=Sad, 4=Surprise, 5=Neutral
        """
        assert num_emotions == 6
        self._num_emotions = num_emotions # Saves the amount of emotion classes

        # Setups the model
        inception_v3 = InceptionV3(
            input_shape=[self._image_size, self._image_size, 3], # The image size the
            network requires
            weights='imagenet', # The pre-trained weights from the ImageNet
            pooling='max', # Uses a max-pooling layer as last pooling layer
```

```

        include_top=False, # Excludes the last layer of the network
    )

    # Unfreezes ALL the layers
    for layer in inception_v3.layers:
        layer.trainable = True

    # Adapts the last layers to classify the six emotion classes
    dense_01 = Dense(units=1024, activation='relu')(inception_v3.output)
    predictions = Dense(units=num_emotions, activation='softmax',
name='predictions')(dense_01)

    self._learning_rate = self._initial_learning_rate # Stores the initial learning
rate

    # Configures the learning process
    model = Model(inputs=inception_v3.input, outputs=predictions)
    model.compile(
to use
        optimizer=Adam(lr=self._learning_rate), # The optimizer and the learning rate

        loss='categorical_crossentropy', # The loss function
        metrics=['accuracy'] # The metrics to calculate
    )
    print(model.summary()) # Prints the model structure

    # Checks if there is a saved model available, if not the new one gets stored
    train_save_path = self._saved_model_dir + "train/"
    files = [file for file in os.listdir(train_save_path)]
    if len(files) is 0:
        model.save(train_save_path + "init.h5")
    return

def create_generator(self, batch_size, mini_batch_type='training', aug=None):
    """
    This method creates a generator to reads the mini-batch data from disk
    and returns them to be feed into the network
    INPUT:
    - batch_size: The batch size to of the images
    - mini_batch_type: Either training or validation
    - aug: If the data should be augmented or not. Has to be None for validation
    IMPORTANT: The data reading process has to be unlimited and only specified by the
epoch number
    """
    index = 0
    while True:
        # Reads the images and labels from the hdf5 file in successive order
        images, labels, index = self.get_successive_batch(index, mini_batch_type)

        # Preprocess the images if necessary
        if images[0].shape[1] != self._image_size or images[0].shape[2] !=
self._image_size:
            images, labels = self.preprocess_images(images, labels)

        one_hot_labels = to_categorical(labels, self._num_emotions)
        # If augmentation is desired
        if aug is not None:
            (images, one_hot_labels) = next(aug.flow(images, one_hot_labels,
batch_size=batch_size))
        if index is -1: # Resets the index to the start again and checks for learning
rate decay
            index = 0

        yield(images, one_hot_labels) # Returns the images and labels to the calling
method

```

```

# CUSTOM CALLBACK-METHOD
def rate_decay(self, epoch):
    """
    This callback method calculates the new learning rate to apply learning rate decay
    INPUT:
    - epoch: The current epoch
    """
    self._learning_rate = self._initial_learning_rate / (
        1 + self._decay_rate * epoch) # Decays the learning rate
    return np.float32(self._learning_rate)

def training(self, augmentation=True, early_stopping=True, decay_rate=-1):
    """
    This method trains the network
    INPUT:
    - augmentation: Flag which determines if the training set gets augmented or not
    - early_stopping: Flag to apply early stopping
    - decay_rate: The learning decay rate, if -1 not learning rate decay gets applied
    """
    print("Training starts...")

    max_train, max_valid, _ = self.get_max_dataset_values() # Gets the amount of
training examples
    train_save_path = self._saved_model_dir + "train/"
    start = datetime.datetime.now()

    # Checks if augmentation is desired
    if augmentation:
        # Creates an ImageDataGenerator which applies augmentation
        aug = ImageDataGenerator(rotation_range=25, zoom_range=0.17,
width_shift_range=0.2,
        height_shift_range=0.2, shear_range=0.15, horizontal_flip=True,
fill_mode="nearest")
    else:
        aug = None

    # Creates the training and validation generator to read the data
    trainGen = self.create_generator(self._mini_batch_size, 'training', aug=aug)
    validGen = self.create_generator(self._mini_batch_size, 'validation', aug=None)

    # Loads a saved model
    files = [file for file in os.listdir(train_save_path)]
    model = load_model(train_save_path + files[0])

    # Setups the callback functions
    callback_list = []
    checkpoint = ModelCheckpoint(train_save_path+'epoch-{epoch:02d}-
val_acc_{val_acc:.2f}.hdf5', monitor='val_acc', verbose=1, save_best_only=True, mode='max')
    # Checks if learning rate decay is desired
    if decay_rate is not -1:
        self._decay_rate = decay_rate
        learning_rate_decay = LearningRateScheduler(self.rate_decay)
        callback_list.append(learning_rate_decay)
    # Checks if early stopping is desired
    if early_stopping:
        early_stop = EarlyStopping(monitor='val_loss', min_delta=0.001, patience=5,
restore_best_weights=True)
        callback_list.append(early_stop)
    callback_list.append(checkpoint)

    # Creates a custom callback function to get the data for the confusion matrix
    predictions_cbk = self.PredictionHistory()
    fetches = [tf.assign(predictions_cbk._true, model.targets[0],
validate_shape=False),

```

```

        tf.assign(predictions_cbk._pred, model.outputs[0],
validate_shape=False)]
        model._function_kwargs = {'fetches': fetches}
        callback_list.append(predictions_cbk)

        # Starts training the network
        history = model.fit_generator(
            trainGen,
            steps_per_epoch=max_train // self._mini_batch_size,
            validation_data=validGen,
            validation_steps=max_valid // self._mini_batch_size,
            epochs=self._max_epochs,
            callbacks=callback_list)

        # PLOTS THE RESULTS
        setup_plot(font_size=22, show_grid=False)
        plot_loss_history(history.history['loss'], history.history['val_loss']) # Plots
the loss history
        plot_accuracy_history(history.history['acc'], history.history['val_acc']) # Plots
the accuracy history
        plot_learning_rate_history(history.history['lr']) # Plots the learning rate
history

        # Plots the confusion matrix
        class_labels = ["Angry/Disgust", "Fear", "Happy", "Sad", "Surprise", "Neutral"]
        plot_names = ["Angry,\nDisgust", "Fear", "Happy", "Sad", "Surprise", "Neutral"]
        target_list = []; prediction_list = []
        # Create the target and prediction lists for the confusion matrix
        for item in predictions_cbk._targets:
            for val in item:
                target_list.append(self.code_to_emotion(val))
        for item in predictions_cbk._predictions:
            for val in item:
                prediction_list.append(self.code_to_emotion(val))
        invalid_labels = [n for n in class_labels if n not in target_list and n not in
prediction_list] # Reads the invalid values
        plot_confusion_matrix(target_list, prediction_list, class_labels, plot_names,
invalid_labels)

        # Prints the result after one epoch
        end = datetime.datetime.now()
        print("Training ends")
        print("Overall duration: " + str(end - start))
        return

def predict(self, image, model):
    """
    This method does the prediction of unknown images
    INPUT:
    - image: The image to predict
    - model: Model to test
    RETURN:
    - The accuracy of the classification step
    """
    if image.shape[0] != self._image_size or image.shape[1] != self._image_size:
        image = self.preprocess_image(image)

    # Executes the prediction
    predictions = model.predict(image)
    percentages = predictions * 100.0
    return percentages[0]

```

```

def testing(self, num_emotions, model):
    """
    This method tests the network
    INPUT:
    - num_emotions: The amount of the emotion classes:
    0=Angry/Disgust, 1=Fear, 2=Happy, 3=Sad, 4=Surprise, 5=Neutral
    """
    assert num_emotions == 6
    self._num_emotions = num_emotions # Stores the number of the emotion classes

    _, _, max_test = self.get_max_dataset_values() # Gets the amount of testing
examples
    train_save_path = self._saved_model_dir + "train/"
    start = datetime.datetime.now()

    # Creates the testing generator to read the data
    testGen = self.create_generator(self._mini_batch_size, 'testing')

    # Loads a saved model
    files = [file for file in os.listdir(train_save_path)]
    model = load_model(train_save_path + files[0])

    # Executes the testing process
    predictions = model.evaluate_generator(
        testGen,
        steps=max_test // self._mini_batch_size,
        verbose=1,
        workers=1)

    end = datetime.datetime.now()
    print("Testing has ended")
    print("Overall duration: " + str(end - start))
    return predictions[1] * 100.0

def code_to_emotion(self, code):
    """
    This method converts the emotion number to the appropriate emotion name
    INPUT:
    - code: The integer code of the emotion
    RETURN:
    - The string name of the respective emotion code, otherwise None
    """
    emotion_names = {0: "Angry/Disgust",
                     1: "Fear",
                     2: "Happy",
                     3: "Sad",
                     4: "Surprise",
                     5: "Neutral"}

    if code >=0 and code <=5:
        return emotion_names.get(code)
    return None

```