- 2. Development phase/reflection phase -

2.1 Framework and libraries

Keras is currently one of the best machine learning frameworks, highly appreciated for its simplicity and efficiency. It is now integrated into TensorFlow, and it contains all necessary tools to create and maintain even the huge and complex tasks. Google Collab was at first chosen as a free cloud based platform that enables decent computation time, combined with google drive as cloud data storage. It is relatively easy to maintain, and does not need any setup or installing. Unfortunately, it has a short time before disconnection and it must be regularly controlled, often crashes before the end of the 16-20 hour session. Second choice is local computing. The program is adapted in PyCharm IDE which needed pre installing of all needed libraries. Numpy is unavoidable when working with arrays and matrices and so is Matplotlib when it comes to visualisation of data in any form including images. Also needed for importing, manipulation and saving the data is OS library, and Random is used for randomised values (picking random image for example)

After installing, they are imported and important variables are set:

```
# use this for 'AI' set
# TRAIN_DIR = "C:/Users/Szilvi/Meine Ablage/Colab Notebooks/AI/train/"
# TEST_DIR = "C:/Users/Szilvi/Meine Ablage/Colab Notebooks/AI/test/"

# use this for 'AI half' set
TRAIN_DIR = "C:/Users/Szilvi/Meine Ablage/Colab Notebooks/AI half/train/"
TEST_DIR = "C:/Users/Szilvi/Meine Ablage/Colab Notebooks/AI half/test/"

# use this for 'AI2' set
# TRAIN_DIR = "C:/Users/Szilvi/Meine Ablage/Colab Notebooks/AI2/train/"
# TEST_DIR = "C:/Users/Szilvi/Meine Ablage/Colab Notebooks/AI2/test/"
# input_size = 208

# size for images
input_size = 48
```

Active data paths for train and test images are assigned to Train_dir and Test_dir variables. Commenting and uncommenting is used to adapt those variables for different sets (three of them).

2.2 Important program segments

Pyplot (plt) library is used to present four random images from every emotion category folder. Program loops within all emotions and for a single emotion pulls out 4 random images:

```
#random choice(emotions)
plt.figure(figsize=(20,15))

for label in range(nr_of_emotions):
      img_folder = TRAIN_DIR + emotions[label]
  for x in range(4):
      # 4 images per emotion
      plt.subplot(len(emotions), 4, label*4+x+1)
      # random image within specific emotion
      random_image = random.choice(os.listdir(img_folder))
      # image drawing with matplotlib
      img = mpimg.imread(img_folder + '/' + random_image)
      plt.imshow(img)
      plt.title(emotions[label])
      plt.axis('off')
plt.show()
```

This gives following result (first two rows shown here):

Img_folder is not hardcoded because there will be several different image collections. All addresses are based on TRAIN_DIR and TEST_DIR variables defined in the beginning, so that whole dependencies can be adjusted by only changing these two. Input_shape and emotions array also alter for different datasets.

The first significant step is normalization and data preparing. Neural networks generally work quicker and easier when values are set between 0 and 1. Images are zoomed by 0.2 or 20 percent because the central part of the image/face contains the majority of significant information anyway. This should increase speed notably without losing accuracy:

```
train_datagen = ImageDataGenerator(rescale = 1./255, zoom_range = 0.2)
```

2.3 Building a CNN model using Keras

Central part of a program for visual recognition is defining the elements of a learning model. At start there where two convolutional layers with 16 and 32 filters standard 3 x 3 size, also standard ReLu activation function followed by batch normalization which increases speed even more, but given poor performing, another one layer is added with 64 matrices:

```
# Add convolutional layers with increasing filter number
model.add(Conv2D(16, kernel_size = (3, 3), activation ='relu', input_shape =
input_shape))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(BatchNormalization())

model.add(Conv2D(32, kernel_size = (3, 3), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(BatchNormalization())

model.add(Conv2D(64, kernel_size = (3, 3), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(BatchNormalization())
```

Pooling reduces the size of the images to 1/4. Without padding, size is additionally 2 px smaller because convolution matrices are 3 by 3. Max pooling used here will pick the biggest value from 4 taken, practically preserving strongest feature traces in the picture with discarding 3/4 of a content. Those two layers (convolution and pooling) with optional addition of a batch normalization are one block for processing. Stacking more of those will explore in greater depth hidden features or dependencies which influence accuracy of prediction. One must only have in mind that the number of images grows through the layers by the factor of number of convolution matrices. One image in the first convolution layer has an output of 16 (convolution matrix processed) images,

those 16 in the next (with 32 matrices) produce 512 images and so on. This is the reason why there were only 2 conv layers in the beginning. Experimenting with 3 layers increased accuracy, but lasted a lot longer.

One feature is used to increase reliability and precision - Dropout.

```
model.add(Dropout(0.25))
```

It randomly excludes some nodes during training, reducing network dependency on features that those nodes contain. It forces the network at that moment to rely on other nodes. This slightly enforces features that would otherwise remain under significance threshold. The whole idea behind this is to find a sweet spot between overfitting and underfitting with given learning material. When a network is trained so, it would extract the majority of characteristics that are truly universal. However, this is an ideal case, there are always limitations in computing power and the images are often very bad or even incorrectly labelled which introduces noise and throws the network in the wrong direction. Dropout value was lowered from initially 0.5 to 0.25 which showed slight improvement.

The model is then compiled with adam optimizer and categorical cross entropy which are common choices for image classification.

```
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

Keras callback which controls some aspect of a training process (in this case accuracy) on epoch end was used to stop training when a certain threshold is reached, but checkpoint callback that saves weights and biases is more useful because of colab often disconnections.

```
# Define the checkpoint callback to save the model every epoch
checkpoint_callback = ModelCheckpoint(checkpoint_filename,
save_best_only=False, save_weights_only=False)
```

Model training epochs were initially set to 100, but it turns out that some maximal validation accuracy is achieved even before 50.

```
# Train the model
history = model.fit(
    train_generator,
    epochs = 50,
    batch_size = 32,
```

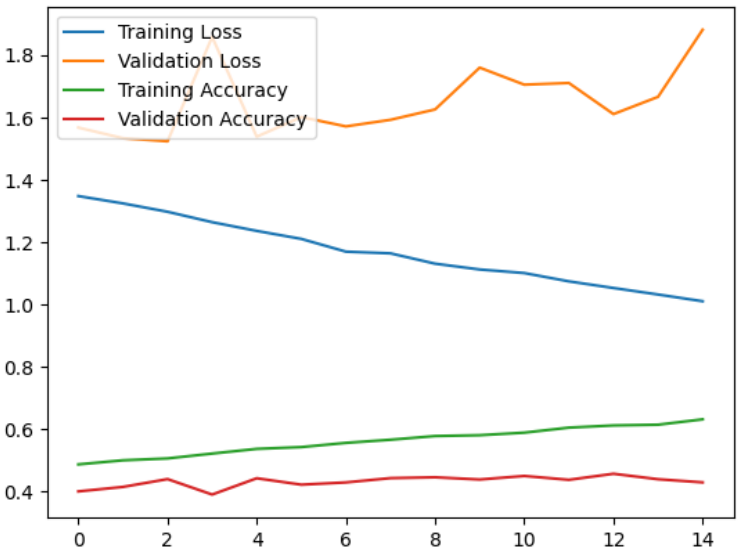
```
    validation_data = test_generator)
```

```
Epoch 36  - accuracy: 0.6771 - val_accuracy: 0.5911
Epoch 43  - accuracy: 0.7043 - val_accuracy: 0.6037
Epoch 50  - accuracy: 0.7246 - val_accuracy: 0.6136
Epoch 68  - accuracy: 0.7667 - val_accuracy: 0.6097
Epoch 92  - accuracy: 0.7956 - val_accuracy: 0.6165
Epoch 100 - accuracy: 0.8015 - val_accuracy: 0.6151
```

It is clear that although training accuracy constantly grows, this has no positive impact on validation accuracy which only oscillates around a certain value.

Dense layers with relu activation function are added, and the last one has the same number of cells as the number of outputs and it is with softmax activation function.

After training process is over, model is saved as model.h5 and important parameters are plotted as visual illustration of model performance:




```
# Plot the loss and accuracy for both the training and validation sets
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.show()
```

Program loads one image that is saved in folder “user images” and also saved model:

```
from keras.models import load_model
import cv2
input_shape = (input_size, input_size, 1)
# Load the saved model
model = load_model(TRAIN_DIR + 'model.h5')

# Load the image
img_path = '/content/drive/MyDrive/Colab
Notebooks/AI/user_images/image1.jpg'
```

This image is preprocessed and prepared for analysing:

```
img = cv2.resize(img, (input_size, input_size))
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # make grayscale

# Reshape the image to match the input shape of the model
img = img.reshape((1,) + img.shape)

# Normalize the image
img = img / 255.0
```

Prediction on image with loaded model is called :

```
prediction = model.predict(img)
```

And finally, predicted label and user image are plotted together:

```
# Display the test image with predicted emotion label
# print (np.shape(img))
plt.imshow(img.reshape(input_size, input_size ,1))
plt.title(predicted_emotion)
plt.axis('off')
plt.show()
```

2.4 Why Convolutional Neural Networks (CNN-s)?

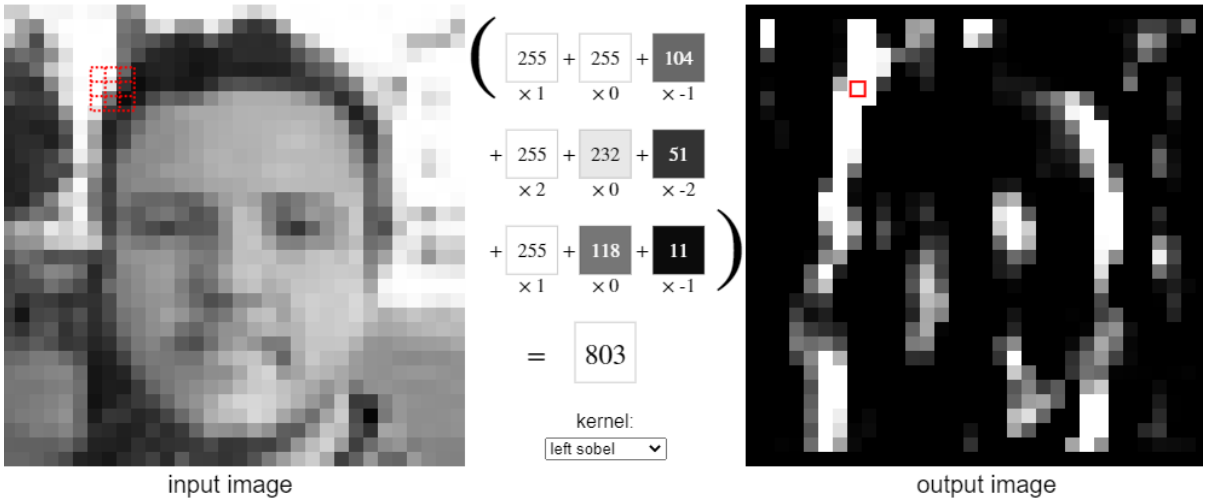
This approach in machine learning is regarded as the best for tasks like image classification. The essence of a technique is exposing an image to one filter that has the ability to extract a unique set of features. It uses a small matrix (usually 3 by 3) whose centre is positioned on the top left corner of the image. Overlapping pixels from matrix and images are multiplied, those results are summed up and this is the value of the first pixel of a resulting image. Example this site shows how it is done:

Matrix “Left Sobel” is applied to the input image:

left sobel

1	0	-1
2	0	-2
1	0	-1

Example input image has 3 by 3 red square on the left side of a person's head, values are shown in the middle where the end result is 803. Program iterates through all pixels in this manner, and on the right is the output image.



input image

output image

Sobel is used (in this case vertical) for line detection and it achieves this by empowering the vertical differences. We see that the middle column has all zeroes, this column will remain neutral in the final outcome. Left and right column sum is zero, meaning when the algorithm reaches a uniform field, the left side of sobel multiplied with one number added to the right side of sobel multiplied with the same number will add up to 0. Thus, uniform surfaces will be demoted. Sharp

vertical line will produce big value and be promoted, aligning distinct vertical features. This principle is used in every filter in CNN layers. Convolution filters are predetermined to pull out a whole range of different features, which will be later encouraged or sunk by the backpropagation. Weights and biases describe what unique combination of thousands of those features and in what proportion means that the observed image belongs to a certain category.

2.5 Data sets

Three Kaggle datasets are taken:

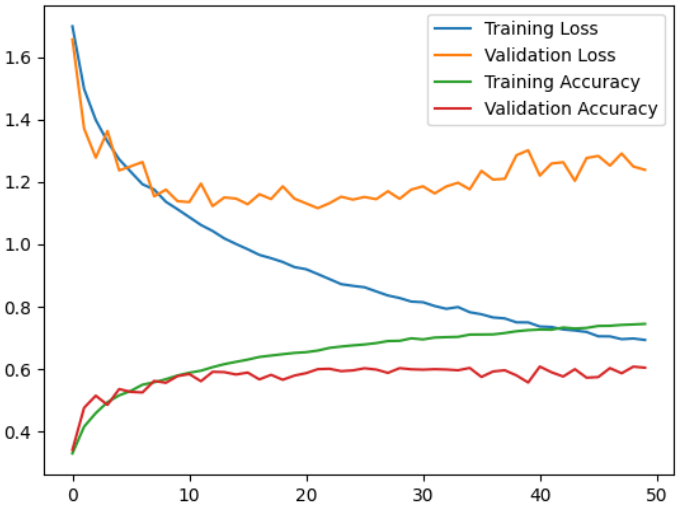
1. Small images with 35900 images, 48 x 48 px (56 MB)
2. Same set with selected 18100 images, 48 x 48 px (34 MB)
3. Bigger images with 14300 images, 416 x 416 px (850 MB)

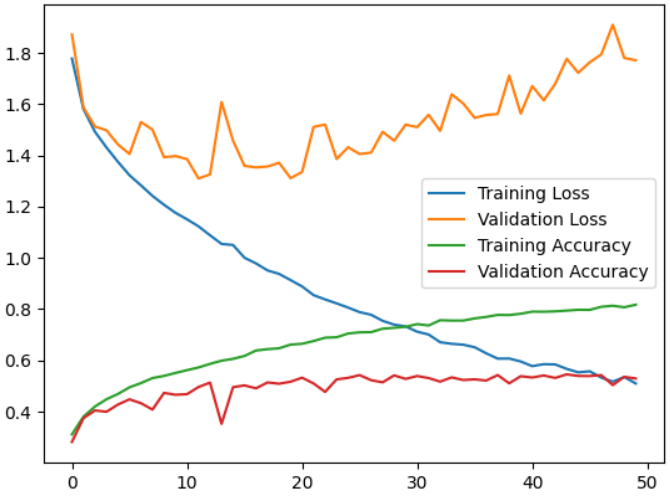
Set with bigger images is chosen because of a reasonable size, and images had some noise and poorly labelled emotion in some cases, so it was relatively close to some real life set. Aldough, labelling of some of the images is really questionable:

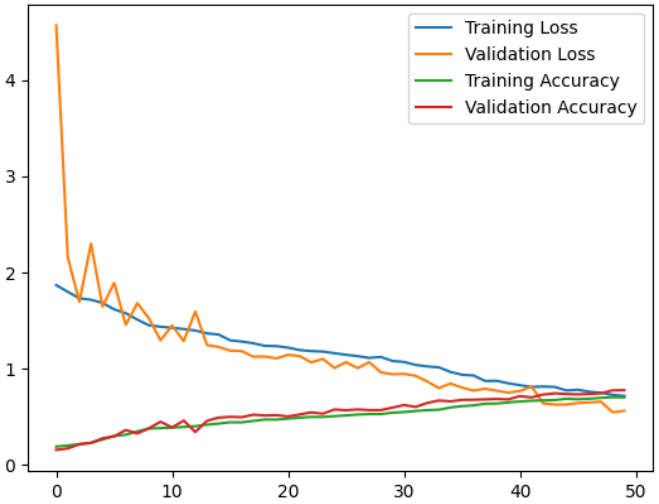
This set is probably made with some small and not quite precise classifier. There are 6 categories: 'Anger', 'Disgust', 'Fear','Happiness', 'Sadness' and 'Surprise'. First and second sets have ‘Neutral’ category added to those. Those two have images size of only 48 x 48 px, and it was interesting to see how much essential information can such small images contain. This set is used for initial program testing and setting because it is quickly processed. Second set would stress out this information because it contains around half images from the first set. All sets have outliers that can’t be clearly classified, but their percent is small.

```
1. set - Epoch 50/50 accuracy: 0.7451 - val_accuracy: 0.6047
2. set - Epoch 50/50 accuracy: 0.8172 - val_accuracy: 0.5292
3. set - Epoch 50/50 accuracy: 0.7036 - val_accuracy: 0.7782
```

Left up: plotted training data set 1., **right up**: plotted training data set 2,
left down: plotted training data set 3., **right down**: example prediction







All conditions depicted as important in ML

-Quality of the data

-Quantity of the data

-Computing power (stacked processing layers which need a lot of computing)

have a profound impact on the quality of trained network. Data with small size has a smaller quantity of usable information that can be extracted in the learning process. Also, accuracy of prediction can't go over 0.6 in case of this neural network configuration or 0,53 in case of one half of a set. This is much bigger than the statistical chance to make a right prediction of 0,143, but it is

far from a usable prediction. Third set is trained on the size of only 208 px (1/2 of original size) because tensorflow reports lack of RAM memory and does not work properly.

2.6 Conclusion

All facts given, this small network design goes in the right direction and manages to actually extract some of the important features that describe emotions. Ofcourse, majority of work is performed by Tensorflow, which supplies Neural Net with proven and quality convolution filters and all needed tools. Tweaking of those tools (for example number of kernels in convolution layer, BatchNormalization or Dropout function) usually change speed or precision of training, but not intensively or more than some 5 - 10%. Number of convolution layers on the other side has improved accuracy by as much as 35 - 40%. Also the quality of data brings completely different outcomes. It was shown that training with a large set of small images gave 18% worse results (60% and 78%). There can be noticed also that a set with half of smaller images (2) has the biggest training accuracy and smallest validation accuracy which implies that this model is overfitted. Third set with big images has the smallest difference between those two parameters, meaning that this outcome is most precise and harmonized.

2.7 Future development

There are images in the third set that are rotated slightly. This technique is useful for adding new images on the walk. Also used are skewing, flipping, cropping which improve stability and robustness of NN. There is a possibility to eliminate them, and instead of that focus on finding face position on image and angle with semantic segmentation, then rotate and crop face to ideal position and use those images to train the network. Same procedure when applying a model - first position and angle detection, then adjustment and finally classification. Yet another network can be used to recognize if there is a usable face on image at all (some images in sets don’t have), which can halt program if there is no any. Also one to separate side photos or those with face covers.

Images of faces have only a handful of those special cases to solve to get a clear, good centred and in any way stabilized face.

This all would allow central NN to be intensively trained for just one task - face emotion recognition, not ballasted with exact same features when face is rotated, skewed or similar. It is similar to learning to read and write in all directions. It could be done, but no real gain in learning 5 or 10 times the same thing, only more possibility for mistakes and confusion.