Artificial Intelligence

Emotion recognition with Convolutional neural networks

Leaded by M.Eng Kazimierz Kiełkowicz

Wojciech Chmura, Konrad Krukar

May 2020

Contents

1 Overview **3**

2 Introduction **3**

3 Convolutional Neural Networks **4**
3.1 Convolutional Layer 4
3.2 ReLU 4
3.3 Pooling layer 4
3.4 Max Pooling 5
3.5 Fully connected layer 5

4 Dataset description **6**

5 Approach 1: Transfer learning with VGG16 **7**
5.1 VGG16 Description 7
5.2 Model building 8
5.3 Data Preprocessing 9
5.4 Code implementation 10
5.5 Results 11

6 Approach 2: Our own CNN model **14**
6.1 Model building 14
6.2 Data Preprocessing 15
6.3 Code implementation 16
6.4 Results 17

7 Conclusion **20**

1 Overview

This project will show the process of building a Convolutional Neural Network to recognize emotions on humans faces.This type of application can be useful and be applied in various systems such as security cameras. As we know nonverbal signals sometimes say more than words.

We take two approaches: pretrained model with our final layers and our own model.

2 Introduction

Humans can use different forms of communications such as speech, hand gestures and emotions. Being able to understand one's emotions and the encoded feelings is an important factor for an appropriate and correct understanding. As such systems that can recognize them, allowing for a more diverse and natural way of communication, are in great demand in many fields. It could for example help during counselling and other healthcare related fields. Other fields like surveillance or driver safety could also profit from it. Being able to detect the mood of the driver could help to detect the level of attention, so that automatic systems can adapt. There are many emotions that can be shown on human faces, but most researchers aim to identify six basic emotions, identified by Paul Ekman - anger, disgust. fear, happiness, sadness and surprise.

Many methods rely on extraction of the facial region. This can be realized in two ways - through manual inference or an automatic detection approach. Methods often involve the Facial Action Coding System which describes the facial expression deconstructing it into the specific action units (AU). An Action Unit is a facial action like "raising the InnerBrow". Multiple activation of AUs can describe the facial expression. Being able to correctly detect AUs is a helpful step, since it allows making a statement about the activation level of the corresponding emotion, but detecting handcrafted facial landmarks can be hard, as the distance between them differs depending on the person. Also, it is significantly harder to determine the facial features of a person when only part of their face is visible or if the lighting conditions are poor.

Our approach uses Convolutional Neural Networks, which is a special kind of ANN and have been shown to work well as feature extractor when using images as input and are real-time capable. This allows for the usage of the raw input images without any pre- or post- processing.

Whole project is inspired by "Accelerating Very Deep ConvolutionalNetworks for Classification and Detection" [3] and "DeXpression: Deep Convolutional Neural" articules [2].

3 Convolutional Neural Networks

A convolutional neural network (CNN) is a class of deep neural networks, most commonly applied to analyzing visual imagery. Convolutional networks were inspired by biological processes - pattern between neurons resembles the organization of the animal visual cortex. A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers, subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers.

3.1 Convolutional Layer

The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable kernels, which have a small receptive field, defined by a width and height. Kernel extends through the full depth of the input volume. During the forward pass, each filter is computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. Stacking the activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. Such a two-dimensional output array from this operation is called a "feature map". Once a feature map is created, we can pass each value in the feature map through a nonlinearity, such as a ReLU, much like we do for the outputs of a fully connected layer.
Let fk be the filter with a kernel size n x m applied to the input x. n x m is the number of input connections each CNN neuron has. The resulting output of the layer calculates as follows:

$$C(x_{u,v}) = \sum_{i=\frac{n}{2}}^{\frac{n}{2}} \sum_{i=\frac{m}{2}}^{\frac{m}{2}} f_k(i,j) x_{u-i,v-j}$$

In summary, we have a input, such as an image of pixel values, and we have a kernel, which is a set of weights, and the kernel is systematically applied to the input data to create a feature map.

3.2 ReLU

ReLU is the abbreviation of rectified linear unit, which applies the non-saturating activation function

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

It effectively removes negative values from an activation map by setting them to zero. It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer.
Other functions could be also used to increase nonlinearity, but ReLU is often preferred to other functions because it trains the neural network several times faster without a significant penalty to generalization accuracy

3.3 Pooling layer

Another important concept of CNNs is pooling, which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling among which max pooling is the most common. It partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75 of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice).
In addition to max pooling, the pooling units can also perform other functions, such as average pooling or even L2-norm pooling. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better

in practice.

3.4 Max Pooling

Max Pooling: Max Pooling reduces the input by applying the maximum function over the input xi. Let m be the size of the filter, then the output calculates as follows: This layer features translational invariance with respect to the filter size.

$$M(x_i) = max\left\{X_{i+k,i+l}||k| \leq \frac{m}{2}, |l| \leq \frac{m}{2}k, l \in \mathbb{N}\right\}$$

3.5 Fully connected layer

The output from the convolution layer was a 2D matrix. Ideally, we would want each row to represent a single input image. In fact, the fully connected layer can only work with 1D data. Hence, the values generated from the previous operation are first converted into a 1D format.

$$\begin{Bmatrix} 9, 32 \\ 14, 26 \end{Bmatrix} \rightarrow \begin{Bmatrix} 9 \\ 32 \\ 14 \\ 26 \end{Bmatrix}$$

Once the data is converted into a 1D array, it is sent to the fully connected layer. All of these individual values are treated as separate features that represent the image. The fully connected layer performs two operations on the incoming data – a linear transformation and a non-linear transformation.

4 Dataset description

Data that we got is in one *.csv [1] file with almost 36 thousands rows and two columns. The data consists of 48×48 pixel gray scale images of faces. The faces have been automatically registered so that the face is more or less centered and occupies about the same amount of space in each image. Columns contain:

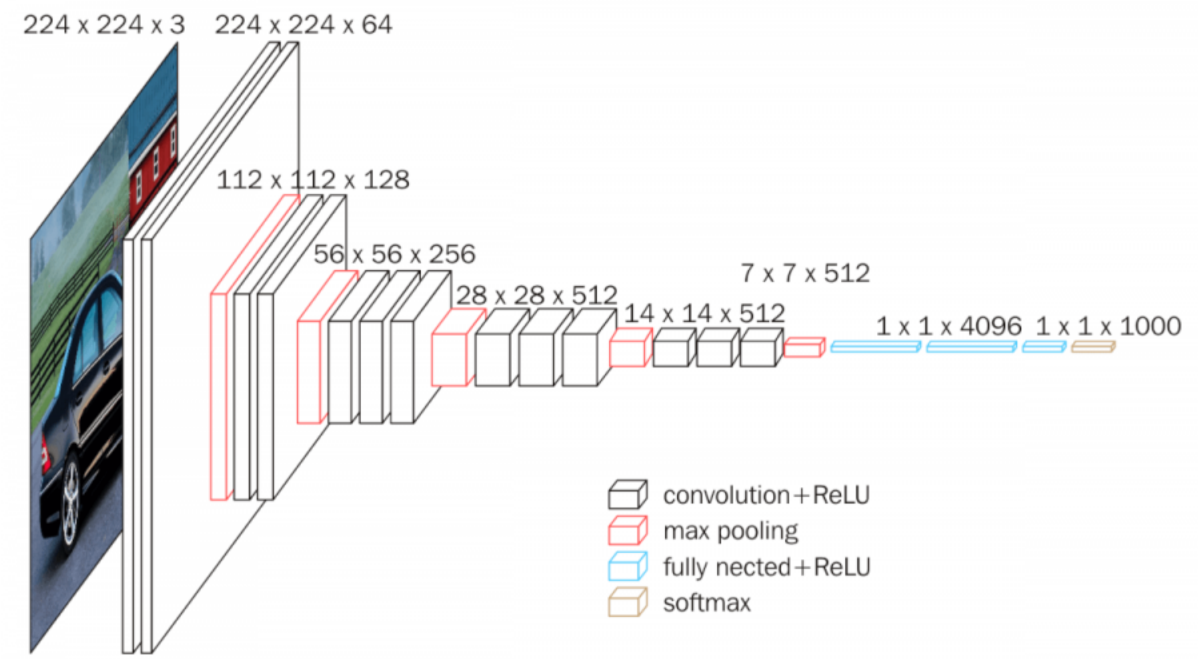
- number of emotion
  in range 0 - 6
  - 0 - angry
  - 1 - disgust
  - 2 - fear
  - 3 - happy
  - 4 - sad
  - 5 - surprise
  - 6 - neutra
- string of pixels
  pixels are in one long string (2304 of them), they are in grayscale 0 - 255.

This pixels will be processed depending on the model.
Training part has 28708 images, testing only 7178.

5 Approach 1: Transfer learning with VGG16

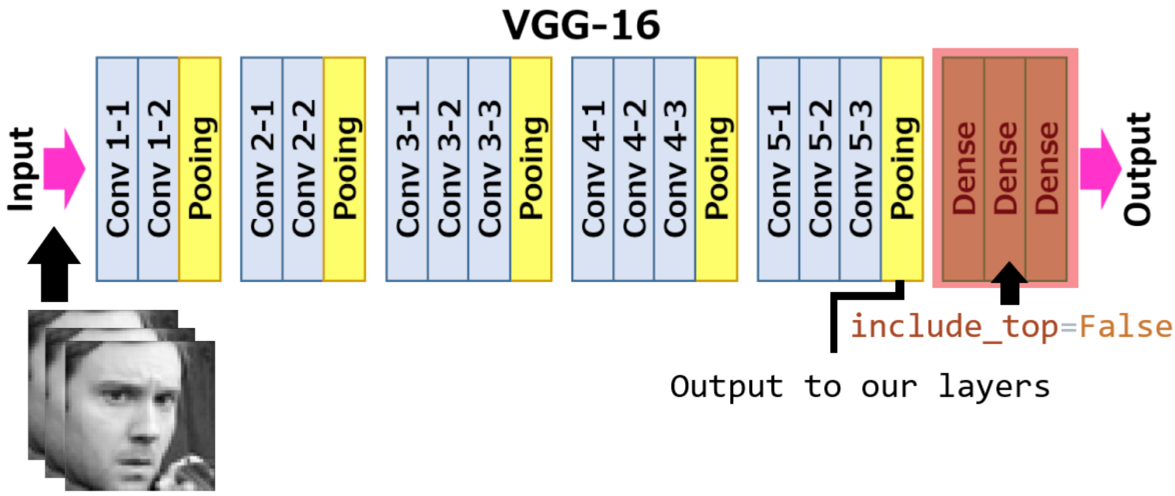
At the beginning we decided to use transfer learning: pretrained VGG16 model with our additional final layers.

5.1 VGG16 Description



VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford for ILSVRC-2014 competition. This model achieves 92.7It is considered to be one of the excellent vision model architecture till date. Most unique thing about VGG16 is that instead of having a large number of hyper-parameter they focused on having convolution layers of very small receptive fields - 3x3 filter with a stride 1. It also always uses same padding and maxpool layers of 2x2 filters of stride 2. It follows this arrangement of convolution and max pool layers consistently throughout the whole architecture. In the end it have three Fully-Connected (FC) layers: the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. All hidden layers are equipped with the rectification (ReLU) non-linearity. It is also noted that none of the networks (except for one) contain Local Response Normalisation (LRN), such normalization does not improve the performance on the ILSVRC dataset, but leads to increased memory consumption and computation time.
The 16 in VGG16 refers to it has 16 layers that have weights. This network is a pretty large network and it has about 138 million parameters.

5.2 Model building



At the beginning we have to implement vgg16 model to our environment:

```
vgg16 = VGG16(include_top=False,
              input_shape=(48, 48, 3),
              pooling='avg',
              weights='imagenet')
```

While working at the model we took two approaches:

- train whole model with vgg16 frozen layers
- get vgg16 output and train our model (final layers) separately

First idea took too much time to train, so we decided to get the output of VGG16 and then proceed through our layers and train only a few layers.
After that we connect this two models: VGG16 + our layers.

5.3 Data Preprocessing

We need to preprocess images into a 48x48 array, so that it could be reorganised as a picture, also we will change the range from 0 -255 to 0 - 1.

```
def processPixels(pixels):
    pixels_list = [item[0] for item in pixels.values.tolist()]

    score_array = []
    for index, item in enumerate(pixels_list):
        data = np.zeros((imageSize, imageSize), dtype=np.uint8)
        pixel_data = item.split()

        for i in range(0, imageSize):
            index = i * imageSize
            data[i] = pixel_data[index:index + imageSize]

        score_array.append(np.array(data))

    score_array = np.array(score_array)
    score_array = score_array.astype('float32') / 255.0
    return score_array
```

After that we need to multiply the layers into (48, 48, 3) because VGG16 CNN is prepared for RGB pictures. So basically recopy gray pixels 2 times more.

```
def makeVGG16input(array_input):
    array_input = np.expand_dims(array_input, axis=3)
    array_input = np.repeat(array_input, 3, axis=3)
    return array_input
```

We have also split this dataset into the training part and testing part.

```
def split_for_test(list):
    train = list[0:TRAIN_END]
    test = list[TEST_START:]
    return train, test
```

5.4 Code implementation

```
# Create VGG16input
x_train_input = makeVGG16input(x_train_matrix)
x_test_input = makeVGG16input(x_test_matrix)

# VGG 16. include_top=False so the output is the 512 and use the learned weights
vgg16 = VGG16(include_top=False,
              input_shape=(48, 48, 3),
              pooling='avg',
              weights='imagenet')

for layer in vgg16.layers:
    layer.trainable = False

x_train_input = vgg16.predict(x_train_input)
x_test_input = vgg16.predict(x_test_input)

# Build and train model
finalLayer = Sequential()
finalLayer.add(Dense(1024, input_shape=(512,), activation='relu'))
finalLayer.add(Dense(128, activation='relu'))
finalLayer.add(Dropout(0.5))
finalLayer.add(Dense(512))
finalLayer.add(Dense(numberOfClasses, activation='softmax'))
finalLayer.summary()

adamax = Adamax()

finalLayer.compile(loss='categorical_crossentropy',
                   optimizer=adamax,
                   metrics=['accuracy'])

# Train
history = finalLayer.fit(
    x_train_input, y_train,
    validation_data=(x_test_input, y_test),
    epochs=300,
    batch_size=50)

# Evaluate
score = finalLayer.evaluate(x_test_input,
                            y_test, batch_size=50)

finalModel = Sequential()
finalModel.add(vgg16)
finalModel.add(finalLayer)
finalModel.compile(loss='categorical_crossentropy',
                   optimizer=adamax,
                   metrics=['accuracy'])
finalModel.save('modelFinal.h5')
```

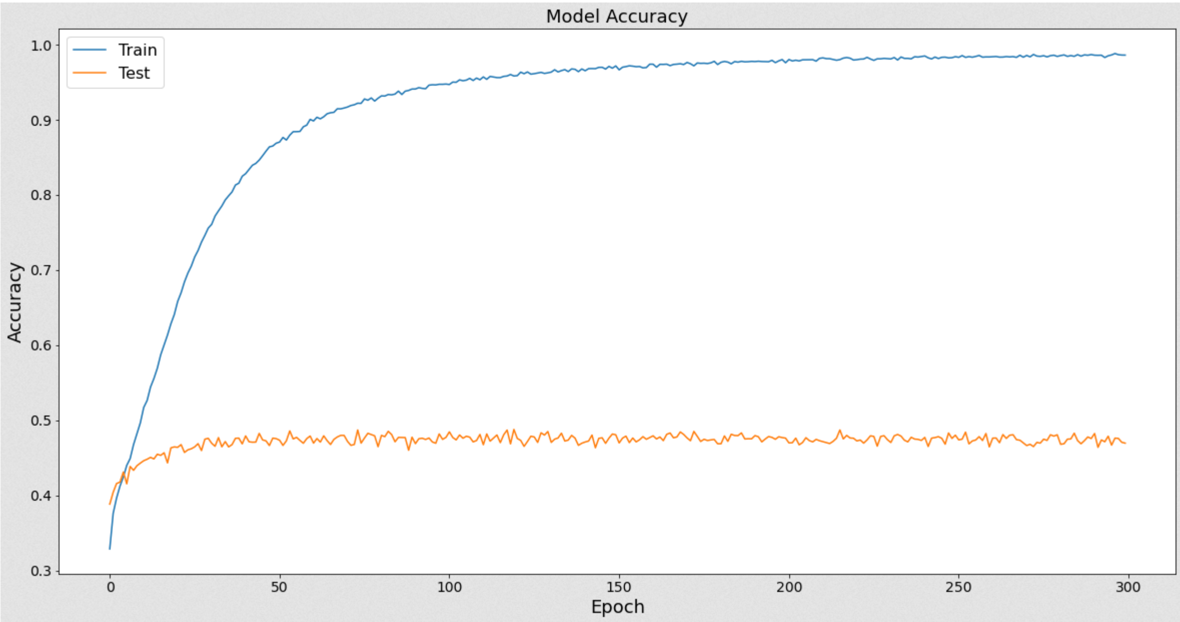
5.5 Results

We have trained our model for 300 epochs with a batch size of 50. Accuracy:

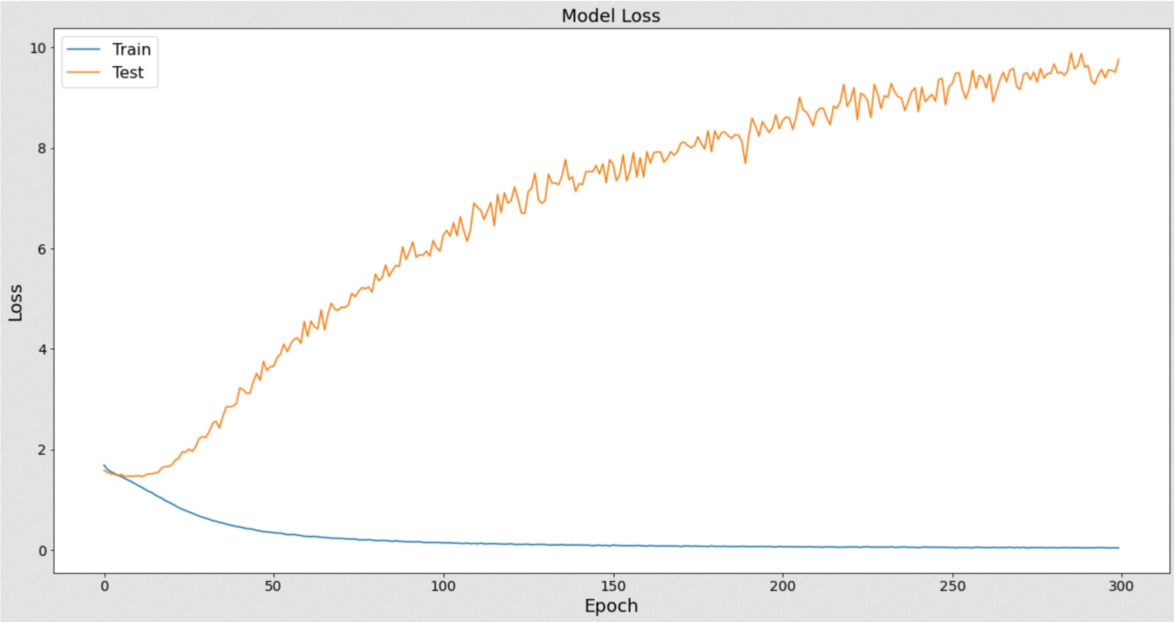
- Training Set:
  - First epochs: 32,8%
  - Last epoch: 98,6%
- Test Set:
  - First epochs: 38,8%
  - Last epoch: 46,9%

```
Epoch 300/300
28708/28708 [==============================] - 2s 72us/step - loss: 0.0398 - accu
7178/7178 [==============================] - 0s 34us/step
After top_layer_model training (test set): [9.76360233835139, 0.469490110874176]
```

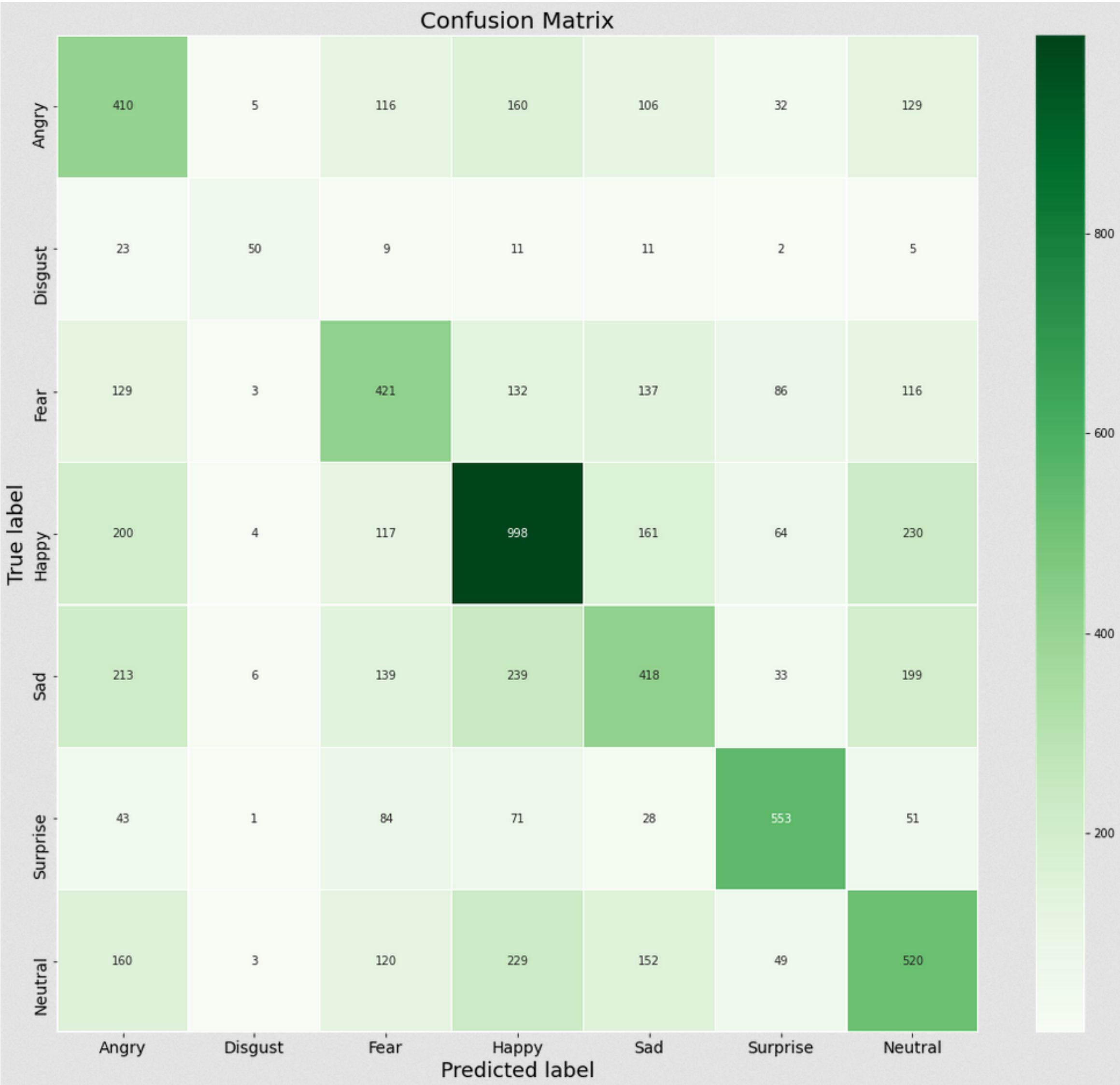
Test/Validation Set above suggest that mode is over-fitting.



As we can, accuracy on the train set is steadily increasing, but on the test set it increases only in the first thirty-forty epochs. Difference in accuracy is probably caused by over-fitting. In real life accuracy will probably be even lower, due to problems like different face position, or bad light. We can do some preprocessing on new images to more closely match the images fed in during training/testing - this should increase accuracy.



Validation loss is much higher than training loss, which proves our theory about over-fitting



Confusion matrix shows us mistakes that model made.

Here we have some examples:

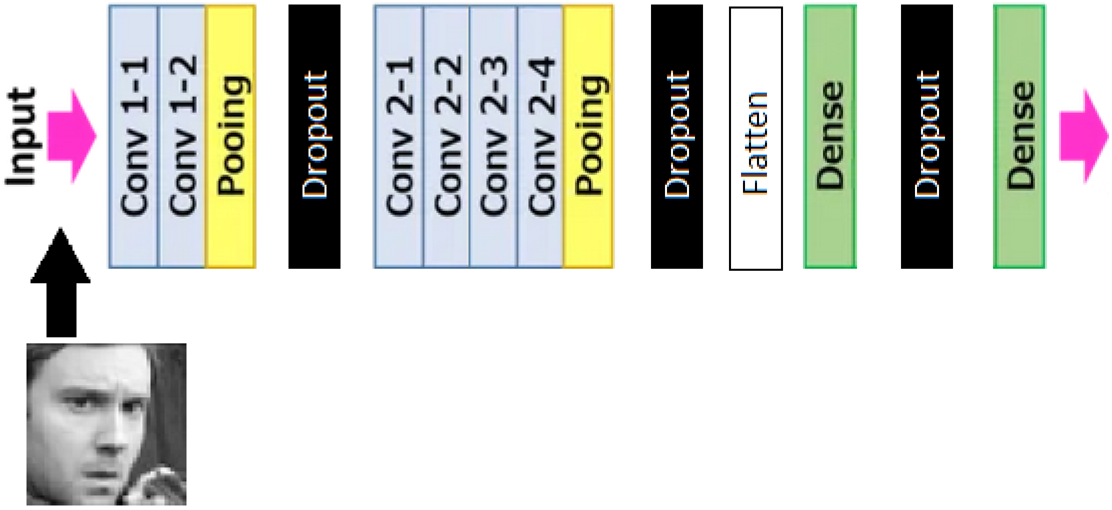
- Correctly predicted images

- Wronly predicted images

6 Approach 2: Our own CNN model

Seeing the results of the previous model, we decided to create our own CNN.

6.1 Model building



Similar to the vgg16 model, we decided that our model will only use very small (3x3) filters in convolutional layers. The input will consist of array with a size of 48 by 48. We started with two layers followed by max Pooling layer with filter size 2x2 and by Dropout layer. Next step is four more convolutional layers, followed by the same two layers are previous step. Then, we flatten output and pass it to two more layers.

It gives us slightly more than 22 million of trainable parameters, our previous model have 15 millions of parameters, most of them pre-trained.

6.2 Data Preprocessing

We need to preprocess images into a 48x48 array, so that it could be reorganised as a picture, also we will change the range from 0 -255 to 0 - 1 and add third dimension.

```
def processPixels(pixels):
    pixels_list = pandas_vector_to_list(pixels)

    score_array = []
    for index, item in enumerate(pixels_list):
        data = np.zeros((imageSize, imageSize), dtype=np.uint8)
        pixel_data = item.split()

        for i in range(0, imageSize):
            index = i * imageSize
            data[i] = pixel_data[index:index + imageSize]

        score_array.append(np.array(data))

    score_array = np.array(score_array)
    score_array = score_array.astype('float32') / 255.0
    score_array = np.expand_dims(score_array, axis=3)
    return score_array
```

We have also split this dataset into the training part and testing part.

```
def split_for_test(list):
    train = list[0:TRAIN_END]
    test = list[TEST_START:]
    return train, test
```

6.3 Code implementation

```
# Build and train model
model = Sequential()
model.add(Conv2D(64, (3, 3), activation='relu', padding='same',
                 input_shape = (48, 48, 1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(256, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(numberOfClasses, activation='softmax'))
model.summary()

adam = Adamax()

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

early_stopping = EarlyStopping(monitor='val_loss', patience=20)

# train
history = model.fit(x_train_matrix, y_train, batch_size=128, epochs=500,
          validation_data=(x_test_matrix, y_test), shuffle=True,
          callbacks=[early_stopping])

# Evaluate
score = model.evaluate(x_test_matrix,
                       y_test, batch_size=50)
```

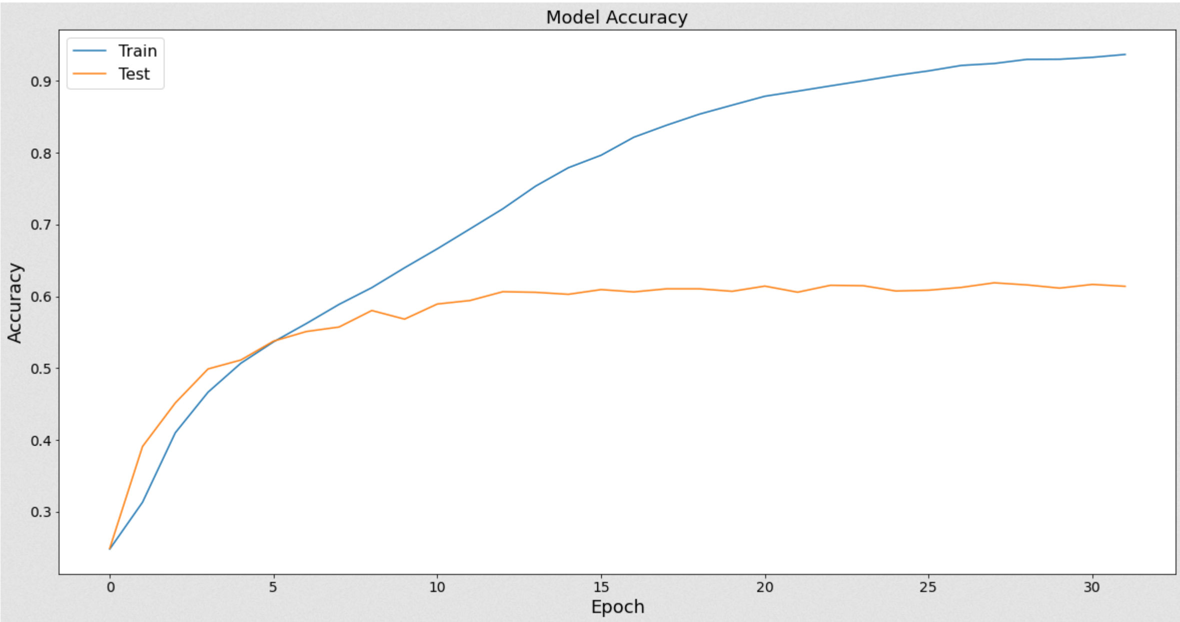
6.4 Results

We have trained our model with a batch size of 128 with. Accuracy:

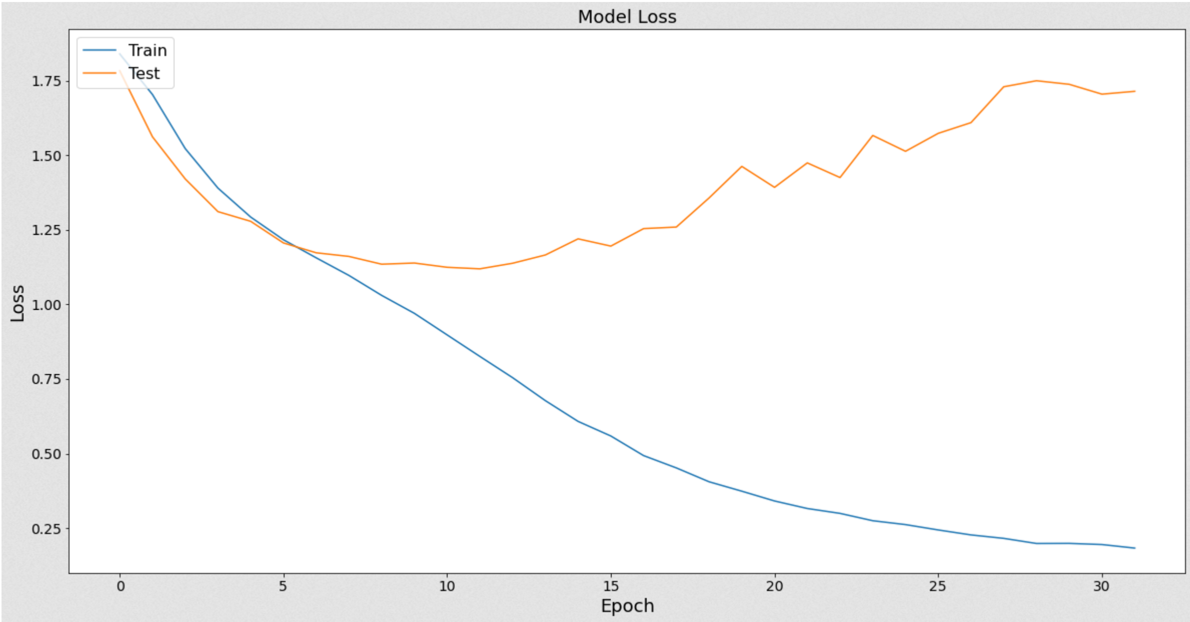
- Training Set:
  - First epochs: 24,8%
  - Last epoch: 93,6%
- Test Set:
  - First epochs: 24,9%
  - Last epoch: 61,3%

```
28708/28708 [==============================] - 23s 788us/step - loss: 0.1830 - accura
7178/7178 [==============================] - 2s 344us/step
After model training (test set): [1.7145241508413471, 0.6139593124389648]
```

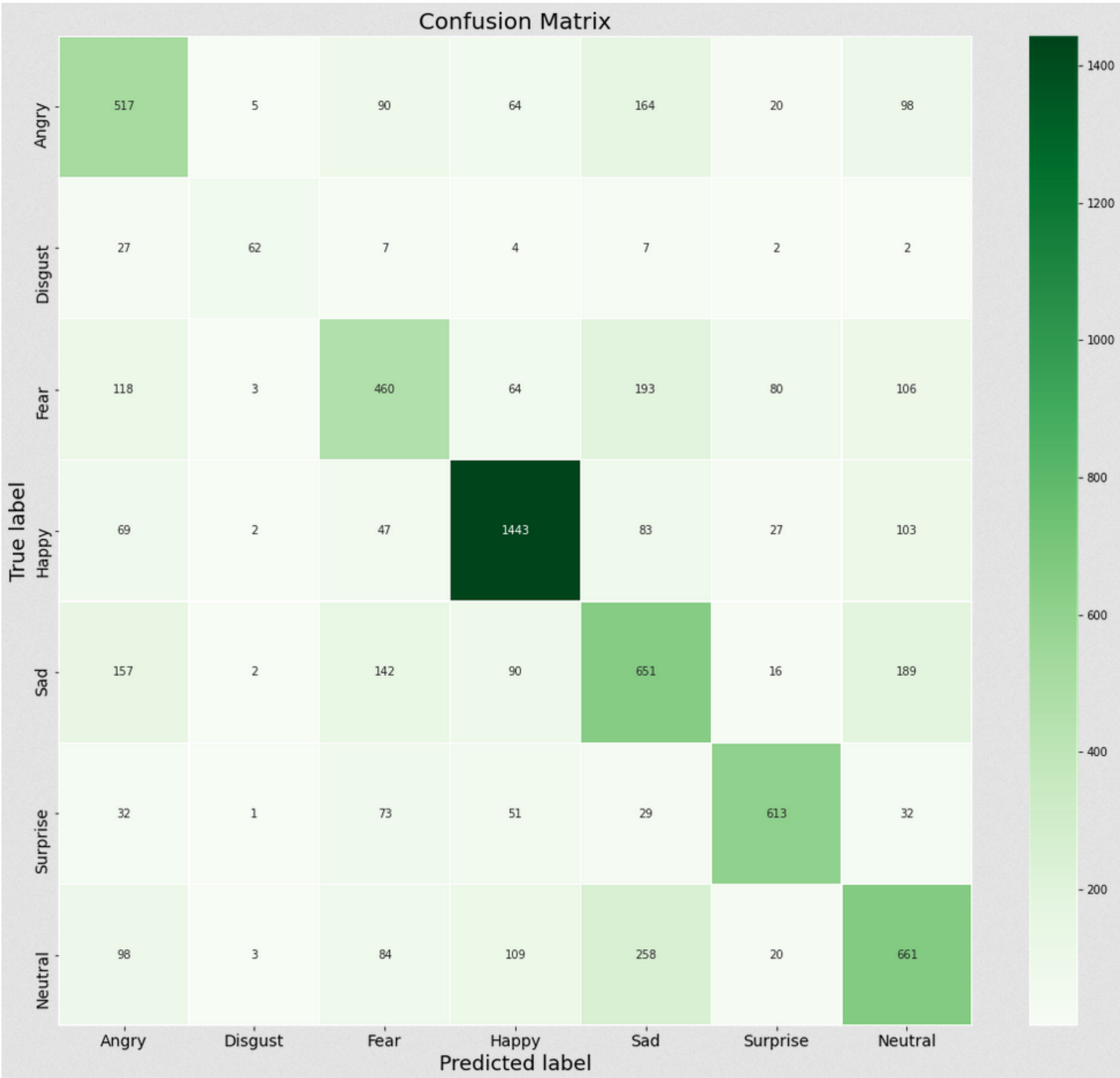
Test/Validation Set result is much batter then in prevoius model.



Accuracy rises as expected.



As we can see model is over-fitting again but much less.



Confusion matrix shows us mistakes that model made.

Here we have some examples:

- Correctly predicted images

- Wronly predicted images

7 Conclusion

Automatic facial recognition has a wide range of applications, such as human-computer interaction and security systems. Being able to understand one's emotions and the encoded feelings is an important factor for an appropriate and correct understanding. With the ongoing research in the field of robotics, especially in the field of humanoid robots, it becomes interesting to integrate these capabilities into machines allowing for a more diverse and natural way of communication. Our models are still far to perfect, but this area of AI will be surely developed in the future.

References

[1] Challenges in representation learning: Facial expression recognition challenge. `https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/overview`, 2013.

[2] Peter Burkert, Felix Trier, Muhammad Zeshan Afzal, Andreas Dengel, and Marcus Liwicki. Dexpression: Deep convolutional neuralnetwork for expression recognition. `https://arxiv.org/pdf/1509.05371.pdf`, 2015.

[3] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutionalnetworks for classification and detection. `https://arxiv.org/pdf/1505.06798.pdf`, 2015.