Python application for Facial Expression Recognition from Live Video

Damir Varešanović

Faculty of Computer and Information Science, University of Ljubljana

Abstract:

This paper reviews the implementation and results of the project assignment regarding facial expression (emotion) recognition using different machine learning and computer vision libraries. In the first part, we describe the data used for training and testing the model. In the next section, we review the Python implementation of the learning and main recognition algorithm. And finally, we will discuss the final results and the conclusion of the assignment.

1. Introduction and motivation

For this year's project assignment of the course "Introduction to Artificial Intelligence" I have chosen to implement a Python application for emotion recognition from live video capture. The main goal of the assignment we are trying to accomplish is whether it is possible to accurately predict person's emotion just from the facial expression. We will be using OpenCV computer vision library for image and video processing, and Keras and Tensorflow libraries for building the supervised machine learning model. Keras is a high-level machine learning api which runs on top of Tensorflow and can access the low level Tensorflow graph.

An example of the usage of this application, would be evaluation of drivers in critical traffic situations. We could observe their facial reactions during those critical situations, combine the results with other biometric data such as heart rate and galvanic skin response and evaluate how calm or nervous the driver actually is during these situations.

2. Dataset

The data is consisted of 35887 labelled grayscale 48x48 pixel images of facial expressions. All images show faces positioned approximately in the centre and occupy the same amount of area. Each facial image represents one of seven emotion expressions: Anger, Disgust, Fear, Happy, Sad, Surprise and Neutral.

The .csv file of the dataset contains three columns. The "emotion" column is consisted of integers ranging from 0 to 6 and labels the above-mentioned emotion expressions (Angry=0, Disgust=1, Fear=2, Happy=3, Sad=4, Surprise=5, Neutral=6). The "pixels" column contains strings of pixel integers separated with spaces. And finally, the third and last "Usage" column contains string identifiers for train and test examples. This column divides the dataset into 28709 examples for training, and 7178 examples for testing.

This dataset can be found and downloaded for free from Kaggle. We could also consider making a separate application for collecting images to create our own dataset. One of the pros of this approach would be that we could get higher quality images and the biggest con is that this approach would take a lot of time, since we would have to manually check and label all collected images. For this assignment, we will use the free dataset from Kaggle, and consider the second option for an upgrade in the future. Three examples from the dataset:

3. Implementation

The assignment was implemented in two Python scripts. The first is used for building and training the supervised machine learning model, and the second one is used as a main script where we capture face images from live video and predict the emotion of the face expression.

I. Building and Training the model

Process of building and training the model is implemented in the "train.py" script and it is functionally divided into 4 tasks.

We start off with data pre-processing. Before we build or even train our model, we have to prepare our data for proper use. The .csv file of the dataset is read into a pandas dataframe, and from there we manipulate each column separately. The "emotion" column is first presented as a regular integer list and additionally converted to a binary matrix representation of each class (example: 0 would presented as [1, 0, 0, 0, 0, 0, 0] since we have 7 different classes). This is important because we will use categorical cross entropy for calculating the loss when training our model. We will discuss this matter again later in this section. The "pixels" column is presented as a numpy array of images. Each image is first converted from a string representation into a two-dimensional numpy array (48x48), and then into a three-dimensional numpy array (48x48x3) so that they fit accordingly to the input of our model. Finally, the "Usage" column is used for dividing the image and class lists into training and testing examples.

Next step is building our model. For the bottom layer of our model (not as an entire model), we are going to use a previously trained neural network alongside pre-trained weights which is provided by the Keras Applications library called VGG16. VGG16 is a convolutional neural network architecture named after the Visual Geometry Group from Oxford, who developed it [1]. The data input shape must be three-dimensional, this is why we converted our images additionally to 48x48x3. We specify that we do not include the top two dense layers of VGG16, and that we want to include the pre-trained weights ("imagenet"). We feed the inputs to the VGG16 and get 512 predictions for each input.

We still haven't built a top layer network which is going to be trained. This part of the network is going to take in the 512 float values which is outputted by VGG16 and try to extract more meaningful data from the input. We achieve this by creating a Sequential network of three fully connected layers and an output layer using Keras. A sequential model is a linear stack of layers, and we are going to create just that [2]. The first layer will input the 512 float values and the fourth layer will output 7 value which represent our final output and prediction. This is possible as every layer dense the input into a smaller more connected output. When we compile our sequential top layer network, we need to specify three parameters: the loss function or the objective that the model will try to minimize (cross entropy, because we need a way to measure the difference between predicted probabilities for each class and groundtruth probabilities [3][4]), metrics (accuracy in our case of classification) and the gradient descent optimizer (Adamax, learning parameters will be discussed in the Results section).

After we created and compiled the top layer network, we can start training it with the outputs from the VGG16 bottom layer network. Besides the training data, we also need to input the number of training epochs and the batch size. Those values are user defined and are inputed in the script as arguments.

When our training is finished, we proceed to merge our bottom VGG16 and top sequential layer networks into a final model object, where the input would be the three-dimensional image array and the output would be the 7 values representing our predictions. We can achieve this by using the Model class API, provided by Keras. Our model is ready to be evaluated on both training and test data to obtain both train and test scores.

Finally, we save our model as a Tensorflow graph, so we can easily access it from our main script and perform emotion recognition from live video capture.

II. Emotion Recognition from Live Video Capture

Process of predicting emotion expressions from live video is implemented in the "main.py" script and it is functionally divided into 2 tasks.

Using OpenCV, we can capture live video from the webcam and obtain frames. Each frame is converted to a grayscale image and using OpenCV's cascade classifier we are able to detect a face in the captured frame. When a face is detected, we simply draw a rectangle around it specifying the region of interest.

By pressing the key "K", the grayscale image captured will be passed to the prediction function. First, we need to reshape the image to a lower resolution of 48x48 and then convert it to our three-dimensional 48x48x3 input shape. We are ready to feed our input image to our model. Our saved model can be opened as a Tensorflow session. Using that session, we can serve the model with the captured image and obtain the probability distribution.

4. Results

We start by analysing the scores based on the user defined input parameters: batch size and number of training epochs.

The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters and the number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset [5].

The first thing we notice in the table below, is that the train scores are significantly higher than the test scores. This is because we evaluated the model on the same data we trained it on. The second thing we notice is that the batch size does not drastically affect the test scores. However, it significantly affects the speed of the training process. With larger batch sizes, internal model parameters are updated less frequently and thus the training process performed faster.

	Train score	Test score

Batch=32
Epochs=500
	99.7%	46.1%

Batch=64
Epochs=500
	99.7%	46.7%

Batch=128
Epochs=500
	99.7%	46.4%

In the table below, we can see that the number of epochs affects the test scores. With larger number of epochs our test scores are lower, and this points to a problem with overfitting.

	Train score	Test score

Batch=32
Epochs=500
	99.7%	46.1%

Batch=32
Epochs=1000
	99.7%	46.5%

Batch=32
Epochs=5000
	99.7%	45.0%

Batch=32
Epochs=10000
	99.8%	43.8%

Finally, we need to analyse the Adamax gradient descent optimizer parameters. In the test scores above, we have used the default recommended parameters (lr=0.002, beta_1=0.9, beta_2=0.999) [6].

In the table below, we also present the scores when the learning rate "lr" is lowered to 0.001.

	Train score	Test score

LR = 0.002
Batch=32
Epochs=500
	99.7%	46.1%

LR = 0.001
Batch=32
Epochs=500
	99.7%	45.2%

5. Conclusion

As we see the results from the previous section, it is safe to say that the assignment was successful. However, when we test the recognition with live video capture we can see that sometimes it has trouble obtaining the correct emotion. This poor performance can be explained by reviewing the limitations with the training data, which are not unsolvable and there is a lot of room for improvement.

The first issue would be that we have to resize high resolution captured images to low resolution, so that it would fit the input shape defined by the dataset images. In this process, we lose the high-resolution details (for example slight smile won't be detected). The same problem occurs when training on these images, micro-expressions will not be "learned" because of the low resolution of train data.

The second issue would be that images from the used dataset, as we mentioned in the second section, are centred and occupy most of the area. This is hard to replicate when capturing face images from live video, and that would explain the poor accuracy of the predictions.

Both issues could be resolved by manually creating a custom dataset with higher resolution images. This would surely be time consuming for both creating the dataset and training the model with larger images, but it would definitely improve the results when predicting emotion from live capture images.

6. References

[1] The Keras Blog – How convolutional neural networks see the world https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html

[2] Keras Documentation - Getting started with the Keras Sequential model https://keras.io/getting-started/sequential-model-guide/

[3] A Friendly Introduction to Cross-Entropy Loss By Rob DiPietro https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/

[4] Cross entropy https://en.wikipedia.org/wiki/Cross_entropy

[5] What is the Difference Between a Batch and an Epoch in a Neural Network? https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

[6] An overview of gradient descent optimization algorithms – Sebastian Ruder http://ruder.io/optimizing-gradient-descent/index.html#adamax