6.835 MiniProject 4: Expression Recognition

Due: 5:00 PM Sunday, March 25, 2018

Figure 1: Actress Scarlet Johansson making various face expressions.

1 Introduction

The goal of this project is to explore the task of expression analysis and emotion classification over two different data sets: 2D images and 3D point clouds. You will implement and compare several neural network architectures, building on what you learned in Mini Project 2. In addition, you'll get experience with Transfer Learning for creating personalized models. Please start early and ask questions!

The data sets you will use in this project are:

1. Kaggle Facial Expression Recognition Challenge [1] image dataset
2. Personal face expression images generated by you
3. Staff generated point clouds from the iPhone X True-Depth Camera

[1] https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data

Included with this project is iMotions's "Facial Expression Analysis: The Complete Pocket Guide." Use this as a reference throughout the project and please go through it before writing any code. It describes the anatomical characteristics of expression and shows examples of each type of expression.

The Facial Action Coding System (FACS) is a tool for measuring expressions first published in 1978 by Ekman and Friesen. It is an anatomical system for describing all observable face movement. It breaks down expressions into individual components of muscle movement known as Activation Units. In this section, we will ask you to describe Facial Expressions based on Activation Units from FACS. A complete guide to FACS and AUs can be found on iMotions blog[2].

2 Getting Started

Download the packaged `.zip` file from Stellar. After extracting the files, open them in your favorite text editor.

Similar to the setup in mini-project 2, use the command prompt to create a virtual machine to hold the dependencies of the mini project. On a Mac, in Terminal you can type:

```
cd MiniProject4/
virtualenv venv -p python3
source venv/bin/activate
```

Instructions for Windows are slightly different[3]:

```
cd MiniProject4/
mkvirtualenv venv -p python3
```

Now the virtual machine should be up and running (you should see the `(venv)` at the front of the terminal line). Note: A virtual environment is not necessary, but is a good habit for developers to follow.

Next, install the dependencies using Pip.

```
pip install -r requirements.txt
```

[2] https://imotions.com/blog/facial-action-coding-system/

[3] See this tutorial for additional info: http://timmyreilly.azurewebsites.net/python-pip-virtualenv-installation-on-windows/

The two provided datasets are located in the `data/` folder. The Kaggle data set is located in `kaggle_fer2013/fer2013.csv`. There are 28K training and 3K testing images in the dataset, each composed of a 48x48 square of pixels and labeled with an emotion [0-6] (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral). The iPhone X dataset is located in `iPhoneX/faces.py`. There are samples of 50 subjects posing with 7 different expressions. Each sample consists of 1220 (x, y, z) points to make up a depth map. We'll explore this data more in Section 5.

This project has 3 main components:

1. Code a Convolutional Neural Network using Keras to detect Face Expressions from a large dataset of images
2. Use transfer learning to adapt the CNN you've created into a personalized network for your own expressions (you'll have to supply the data)
3. Explore a new type of available face data - point clouds from the iPhone X True Depth Camera and classify samples by modifying your original CNN.

Please do not hesitate to post for help on Piazza using the `mini_project_4` tag.

3 Expression Recognition with Convolutional Neural Networks

As you read about and saw during the lecture on face recognition, the hidden layers of a Convolutional Neural Network (CNN) typically consist of some combination of convolutional layers, pooling layers, fully connected layers and normalization layers. We'll give a brief overview of the CNNs and what these layers do. If you don't have any experience with CNNs, watch this video[4] on YouTube to get a solid understanding of how they function.

Together, we'll code up a simple CNN to process the Kaggle dataset. You are encouraged to improve this model by adding additional layers. The Input layer will take in image data (represented as a matrix of numbers) and pass them into a convolution layer. Here the image data is "convolved" - a filter (i.e., a function) is applied methodically to overlapping tiles of the input. The values the the filter produces (technically, the dot product of the filter with each sub matrix, is itself another matrix of data. The final layer, the fully connected layer, takes the convolution and produces a vector of predictions.

[4] Convolutional Neural Networks (CNNs) explained: https://www.youtube.com/watch?v=YRhxdVk_sIs

Note: One epoch while training your model on the full Kaggle dataset can take up to 10 minutes on a laptop, depending on the complexity of your model and the power of your laptop.

Now it's time to write a CNN using the Keras API and Tensorflow backend. We've already started an implementation for you in `cnn.py`. You should complete the implementation by following these steps:

1. We need to understand our input before we can begin our model. The Keggle dataset contains 35888 images: 28709 for training and 3589 for testing. Let's organize this data so we can use it in out model.
   (a) The supplied code imports the data from the `.csv` file for you. Each line of data contains an *emotion label*, *image data*, and *test/train usage*. The *emotion label* is a number between 0 and 6, corresponding to the labels specified above. We parse the `.csv` file into *x_train*, the training image pixel data as 1D arrays of pixels, *y_train*, the labels corresponding to the training images, *x_test*, the testing image pixel data as 1D arrays of pixels, *y_test*, the labels corresponding to the test images.
   (b) The pixel values in the *image data* are strings. Convert them to float32 and normalize the inputs to be between 0 and 1.
   (c) Reshape the image data so we can enter samples into our modes with the shape (48, 48, 1).
2. Now it's time to code the CNN. We'll make use of the Keras functional API for building models. For more information on Keras, see the Keras Tutorial online and explore the documentation here[5]. We'll create a simple model together that gives you a working solution. However, we will later expect you to add additional layers to this model to improve its performance.
   (a) Create an Input layer that takes in data of shape (48, 48, 1,). This is the size of our photos
   (b) Add a Convolutional layer to the network using the 2D convolution layer for spatial convolution over images. Make sure the layer has the following properties: `filters=64.`, `kernel_size=(5,5)`, `activation='relu'`
   (c) Add a MaxPooling2D layer with `pool_size=(5,5)` and `strides=(2, 2)`
   (d) Add a `Flatten` layer which converts the data into a 1D feature vector ready for classification
   (e) Add a `Dense` layer with 1024 units and `activation='relu'`
   (f) Add a final `Dense` layer with 7 units (for classification) and the 'softmax' activation function

[5] https://keras.io/getting-started/functional-api-guide/

3. Now it's time to train our model and see how well it performs.

   (a) Batch the training and testing data using the Keras `ImageDataGenerator()` with `.flow(..., batch_size = 256)`. The `ImageDataGenerator` does image augmentation and artificially creates training images through different ways of processing or combination of multiple processing, such as random rotation, shifts, shear and flips, etc. Here we are using it to randomly selected training set instances for our model.

   (b) Compile your model with `loss='categorical_crossentropy'` and the Adam optimizer (i.e. `keras.optimizers.Adam()`).

   (c) Train your model by calling `model.fit_generator(...)` and the provided $steps\_per\_epoch = len(x\_train)/batch\_size$ and $epochs = 5$ variables. Make sure to save your model after training it: export the model to a `.h5` file using the built in `model.save('model_1.h5')` (please use this naming convention). The model should take about 10 minutes to run and should achieve about 55% accuracy.

   (d) You can test your model with the `test_cnn.py` script to see how a certain example gets classified.

4. Now, modify the above model to improve your accuracy. You may change the parameters (such as *batch_size*) and layers of the model. Draw a diagram of your final network architecture. Describe the structure and the parameters you used. Report the accuracy and loss for your model. In the original Kaggle challenge, the winner achieved just 34% accuracy - so congrats, your model is already much better! Be sure to save your trained model as `model_2.h5`, i.e., followng our naming convention.

4 Transfer Learning for a Personalized Machine Learning Model

In practice, very few people train an entire Convolutional Network from scratch (with random initialization) as we just did. This is because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a CNN on a (different) very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the CNN either as an initialization or a fixed feature extractor for the task of interest.

The process of sharing results across different problems is known as Transfer Learning. In other words, Transfer Learning is a machine learning technique where a model trained on one task is re-purposed for a second related task.

In the section, we'll first ask you to generate your own expression data (yes, you'll have to take pictures of yourself) and use that data to fine-tune your

CNN from the previous section into a personalized model for your face, using Transfer Learning.

5. Pose in front of a camera or get a friend to help you take 7 pictures: one for each type of expression: angry, disgust, fear, happy, sad, surprise, neutral. This is your training data. Crop these images so that they contain a bounding box of only your face. It is OK if the pictures are not 48 x 48, Keras will resize them.
6. Take 7 more pictures. This is your testing data.
7. Now it's time to load your model from the previous section. We've started an implementation for you inside of `transfer.py`
   (a) Add your test and train images to a new folder in the `data/` folder
   (b) In `transfer.py`, import the data from your images and reshape the data so that you can retrain your model from Section 3 (`model_2.h5`). You will need to grayscale your images. You may find functions in `keras.preprocessing` useful for image manipulation.
   (c) Load your model using `load_model('model_2.h5')` and train the model on your 7 training images.
   (d) Finally, test the newly trained model on your test images. We've included a helper function, `plot_emotion_prediction(pred)` that takes in a model prediction values from a single call of `model.predict(x)` and plots them on a bar graph. Which expressions are not being recognized? Why do you think some expressions are recognized better than others? Report the accuracy of the model.
   (e) If your model did not achieve good accuracy on your personal data, explain why you think that is.
   (f) Save your trained model as `model_3.h5`.

5 From Images to Depth: Next Generation of Face Representation

In this section, we expect you to implement a fully functional model on your own. Your work in this part will be graded on correctness, not on how accurate your final model is. You should write your code in `cnnX.py`.

Recall structured light: the technology used by the Xbox Kinect that shines thousands of infrared dots in an area and can create a corresponding depth map. The iPhone X also uses this technology to scan a user's face. We've accumulated samples of 100 iPhone X users posing with different face expressions

using Apple's ARKit framework[6]. Apple anchors the face into a specific origin, and provides us with vertex positions for each point in the face mesh. These vertices are referred to as a *point cloud*. The point cloud we receive is sparse, so we see a smoothed version of an actual face. Together, this normalizes all of our data and makes it ready for analysis.

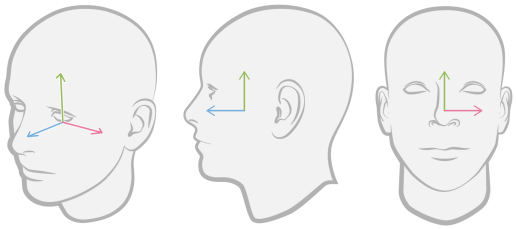

Figure 2: Origin of the face coordinate system.

In this last part, we'll classify the 3D data into the same 7 emotions: angry, disgust, fear, happy, sad, surprise, neutral. However, this time we're expecting you to craft the model. The model you will create should be a 3D Convolutional Neural Network. They are the same in essence to 2D CNNs, but perform operations in 3 dimensions. In order to use a 3D CNN, we'll have to transform our input data into *voxels*. Voxels are the three-dimensional analogue of a pixel: unit volumes of space that contain a value.

9. Visualize the various expressions, use the Visualization GUI by running:

   ```
   >> python show_gui.py
   ```

   You'll be able to see the 7 different expressions. You can drag the graph to view the data from different orientations. Inspect the 3D face data and give us your impressions. Compare the expression data using FACS. Which expressions are the most unique? Which expressions are most similar? What information does the point cloud provide us that the image does not?

10. The data is provided as a Python dictionary `face_samples = { sample id : { emotion: { x: [...], y: [...], z: [...]}}}`. For each point cloud, create a 24 x 24 x 24 voxel grid represented as a 3D numpy array initialized with all 0s. For each point in the cloud, increment the value of the voxel that the point falls in.

11. Construct a 3D Convolution Neural Network using Keras. You can use your previous work as a starting point, but will have to make use of `Conv3D` and `MaxPool3D` from Keras. You are free to add as many layers as you'd like.

[6]Specifically we use this API: https://developer.apple.com/documentation/arkit/arfaceanchor?language=objc

12. Train and test your model. Draw and describe your model architecture. Report your test/train ratio, your batch size, number of epochs, accuracy and loss.

13. Save your trained model as `model_4.h5`.

Extra Credit 1 (5 points): Continue to modify your model until you achieve an accuracy >= 42%. Save your model as `model_ec1.h5`.

Extra Credit 2 (10 points maximum): Use a different classification technique to classify the data (it does not have to be deep learning). Describe what you built and report how well your classification works. Be sure to include where to find your code and instructions for how to run it.

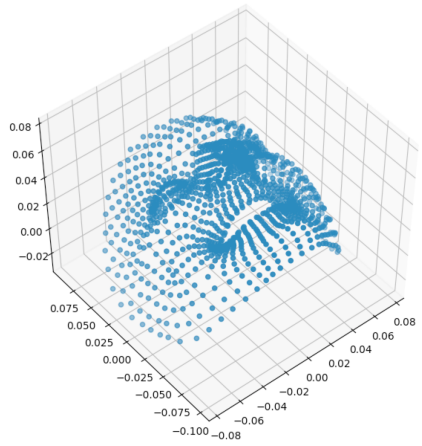

Figure 3: "Happy" expression generated by the iPhone X

6 Submission

You should submit to Stellar a zip file containing the following items:

1. A folder containing all the files needed to run your code (including all supporting files given as part of the starter code). **Note: DO NOT include the `venv/` or `data/` folders in your submission.**

2. Your 4 saved trained models (from Sections 3, 4, and 5) in `.h5` format.

3. A folder containing the 14 images of yourself making the 7 expressions.

4. A PDF of your write-up containing all of your answers to the questions in this assignment (including your network architecture diagrams). If you completed any extra credit, write about which extra credit you did and how you did it.