Facial expression detection using Machine Learning

Introduction

Facial expressions are used by human beings to convey their emotions and feelings without saying anything. Nowadays, Facial Expression Detection has got its applications in many fields such as Robotics, Driver Fatigue Monitoring Systems, Medical treatment and many other human-computer interaction systems. In this article, facial expression detection of the most important and universal facial expressions are illustrated using machine learning.

Machine Learning Problem Formulation

We will use Tensorflow to build Convolutional Neural Network model for conducting image classification. CNN is a class of deep neural networks which is used mainly in computer vision. Our main objective here is to categorize each image on the basis of the emotion shown into one of the seven classes (0 for Angry, 1 for Disgust, 2 for Fear, 3 for Happy, 4 for Sad, 5 for Surprise, 6 for Neutral).

Data Set Source

The data set can be downloaded through the following link:

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

The dataset consists of 48x48 pixel grayscale images of face expressions. The images have been taken such that the face is center aligned and occupies about the same amount of space in each image.

Prerequisites

Software and important libraries needed are Python 3, Jupyter Notebook, numpy, pandas, matplotlib, scikit-learn and seaborn.

Working Code:

Step 1: Importing the necessary libraries.

```
import pandas as pd
import numpy as np
import tensorflow as tf
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.cm as cm
```

Step 2: Now we read the data and get relevant information regarding the data.

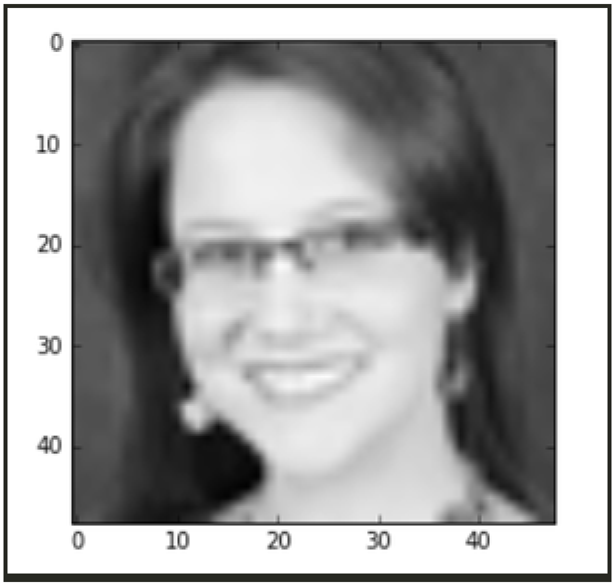
```
data = pd.read_csv("/home/Desktop/fer2013/fer2013.csv")
#check the number of images and each image data variable
data.shape
data.head()
np.unique(data["Usage"].values.ravel())
train_data = data[data.Usage == "Training"]
pixels_values = train_data.pixels.str.split(" ").tolist()
pixels_values = pd.DataFrame(pixels_values, dtype=int)
images = pixels_values.values
images = images.astype(np.float)
```

The following function is used to to show image through 48*48 pixels

```
def show(img):
    show_image = img.reshape(48,48)
    plt.imshow(show_image, cmap='gray')
```

For example, we show the following image.

```
show(images[7])
```



Step 3: Now preprocessing of data has to be done which involves converting flattened data to 48*48 matrix. In data preprocessing, we transform the data into a useful format.

```
images = images - images.mean(axis=1).reshape(-1,1)
images = np.multiply(images,100.0/255.0)
each_pixel_mean = images.mean(axis=0)
each_pixel_std = np.std(images, axis=0)
images = np.divide(np.subtract(images,each_pixel_mean), each_pixel_std)
image_pixels = images.shape[1]
print 'Flat pixel values is %d'%(image_pixels)
image_width = image_height =
np.ceil(np.sqrt(image_pixels)).astype(np.uint8)
labels_flat = train_data["emotion"].values.ravel()

labels_count = np.unique(labels_flat).shape[0]
```

To get one hot encoding outputs, we convert the dense format to one hot encoding format. One hot encoding is basically used to represent categorical values as binary vectors.

```
def dense_to_one_hot(labels_dense, num_classes):
    num_labels = labels_dense.shape[0]
    index_offset = np.arange(num_labels) * num_classes
    labels_one_hot = np.zeros((num_labels, num_classes))
    labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
    return labels_one_hot
labels = dense_to_one_hot(labels_flat, labels_count)
```

```
labels = labels.astype(np.uint8)
```

We now split the data into training & validation data sets.

```
VALIDATION_SIZE = 1709
validation_images = images[:VALIDATION_SIZE]
validation_labels = labels[:VALIDATION_SIZE]

train_images = images[VALIDATION_SIZE:]

train_labels = labels[VALIDATION_SIZE:]
```

Step 4: This is the most important step where we build the Tensorflow CNN Model.

First of all we initialize the weights.

```
def weight_variable(shape):
    initial = tf.compat.v1.truncated_normal(shape, stddev=1e-4)
    return tf.Variable(initial)
def bias_variable(shape):
    initial = tf.compat.v1.constant(0.1, shape=shape)

    return tf.Variable(initial)
```

Now we define the functions *conv2d* and *max_pool_2x2* for convolution and pooling. Convolution is used to extract features from an input image. Pooling is basically used to reduce the dimensions of the data where the combination of clusters of neurons at one layer merge to form a single neuron in the successive layer.

```
def conv2d(x, W, padd):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding=padd)
def max_pool_2x2(x):

    return tf.nn.max_pool(x, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
padding='SAME')
```

We define the input & output of Neural Network.

```
tf.compat.v1.disable_eager_execution()
# images
x = tf.compat.v1.placeholder(tf.float32, shape=(None, image_pixels))
# labels
y_ = tf.compat.v1.placeholder(tf.float32, shape=(None, labels_count))
```

We define the first convolutional layer which is the building block for a CNN.

```
W_conv1 = weight_variable([5, 5, 1, 64])
b_conv1 = bias_variable([64])
# (27000, 2304) => (27000,48,48,1)
image = tf.reshape(x, [-1,image_width , image_height,1])
#print (image.get_shape()) # =>(27000,48,48,1)
h_conv1 = tf.nn.relu(conv2d(image, W_conv1, "SAME") + b_conv1)
#print (h_conv1.get_shape()) # => (27000,48,48,64)
h_pool1 = max_pool_2x2(h_conv1)
#print (h_pool1.get_shape()) # => (27000,24,24,1)
h_norm1 = tf.nn.lrn(h_pool1, 4, bias=1.0, alpha=0.001/9.0, beta=0.75)
```

Now we define the second convolutional layer.

```
W_conv2 = weight_variable([5, 5, 64, 128])
b_conv2 = bias_variable([128])
h_conv2 = tf.nn.relu(conv2d(h_norm1, W_conv2, "SAME") + b_conv2)
#print (h_conv2.get_shape()) # => (27000,24,24,128)
h_norm2 = tf.nn.lrn(h_conv2, 4, bias=1.0, alpha=0.001/9.0, beta=0.75)
h_pool2 = max_pool_2x2(h_norm2)
```

The local layer weight initialization is done using the following functions.

```
def local_weight_variable(shape):
    initial = tf.compat.v1.truncated_normal(shape, stddev=0.04)
```

```
    return tf.Variable(initial)
def local_bias_variable(shape):
    initial = tf.compat.v1.constant(0.0, shape=shape)
    return tf.Variable(initial)
```

The densely connected layer local 3 is added which gives relevant features from all the combinations of features of the previous layer.

```
W_fc1 = local_weight_variable([12 * 12 * 128, 3072])
b_fc1 = local_bias_variable([3072])
# (27000, 12, 12, 128) => (27000, 12 * 12 * 128)
h_pool2_flat = tf.reshape(h_pool2, [-1, 12 * 12 * 128])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
#print (h_fc1.get_shape()) # => (27000, 1024)
```

We now add another densely connected layer.

```
W_fc2 = local_weight_variable([3072, 1536])
b_fc2 = local_bias_variable([1536])
# (40000, 7, 7, 64) => (40000, 3136)
h_fc2_flat = tf.reshape(h_fc1, [-1, 3072])
h_fc2 = tf.nn.relu(tf.matmul(h_fc2_flat, W_fc2) + b_fc2)
#print (h_fc1.get_shape()) # => (40000, 1024)
```

Now the dropout layer is added. It is used to avoid overfitting as it sets input units to 0 with a frequency rate at each step during training of data.

```
keep_prob = tf.compat.v1.placeholder('float')
h_fc2_drop = tf.nn.dropout(h_fc2, keep_prob)
```

We define the readout layer for deep network.

```
W_fc3 = weight_variable([1536, labels_count])
b_fc3 = bias_variable([labels_count])
y = tf.nn.softmax(tf.matmul(h_fc2_drop, W_fc3) + b_fc3)
#print (y.get_shape()) # => (40000, 10)
```

Now, we set the learning rate which is a hyperparameter which shows how quickly the model adjusts to the given problem.

```
LEARNING_RATE = 1e-4
```

The cost function, optimization function, evaluation and prediction function are defined below.

```
# cost function
cross_entropy = -tf.reduce_sum(y_*tf.compat.v1.log(y))
# optimisation function
train_step =
tf.compat.v1.train.AdamOptimizer(LEARNING_RATE).minimize(cross_entropy)
# evaluation
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, 'float'))
# prediction function
#[0.1, 0.9, 0.2, 0.1, 0.1 0.3, 0.5, 0.1, 0.2, 0.3] => 1
predict = tf.argmax(y,1)
```

Now, we set the training iterations to 3000.

```
TRAINING_ITERATIONS = 3000
DROPOUT = 0.5
BATCH_SIZE = 50
```

The following function serves the data by batches. When all training data gets used, it is reordered randomly.

```
epochs_completed = 0
index_in_epoch = 0
num_examples = train_images.shape[0]

# serve data by batches
def next_batch(batch_size):
    global train_images
    global train_labels
    global index_in_epoch
    global epochs_completed
    start = index_in_epoch
    index_in_epoch += batch_size
    # when all trainig data have been already used, it is reorder randomly
    if index_in_epoch > num_examples:
        # finished epoch
        epochs_completed += 1
        # shuffle the data
        perm = np.arange(num_examples)
        np.random.shuffle(perm)
        train_images = train_images[perm]
        train_labels = train_labels[perm]
        # start next epoch
        start = 0
        index_in_epoch = batch_size
        assert batch_size <= num_examples
    end = index_in_epoch
    return train_images[start:end], train_labels[start:end]
```

We now start the TensorFlow session and initialize visualization variables.

```
# start TensorFlow session
init =  tf.compat.v1.initialize_all_variables()
sess =  tf.compat.v1.InteractiveSession()
sess.run(init)
# visualisation variables
train_accuracies = []
validation_accuracies = []
x_range = []
display_step=1
```

The following function checks progress on every 1st,2nd,...,10th,20th,...,100th... step and prints training and validation accuracy for each step.

```
for i in range(TRAINING_ITERATIONS):
    #get new batch
    batch_xs, batch_ys = next_batch(BATCH_SIZE)
      # check progress on every 1st,2nd,...,10th,20th,...,100th... step
    if i%display_step == 0 or (i+1) == TRAINING_ITERATIONS:
        train_accuracy = accuracy.eval(feed_dict={x:batch_xs,
                                                  y_: batch_ys,
                                                  keep_prob: 1.0})
        if(VALIDATION_SIZE):
            validation_accuracy = accuracy.eval(feed_dict={ x:
validation_images[0:BATCH_SIZE],
                                                            y_:
validation_labels[0:BATCH_SIZE],
                                                            keep_prob:
1.0})
            print('training_accuracy / validation_accuracy => %.2f / %.2f
for step %d'%(train_accuracy, validation_accuracy, i))
            validation_accuracies.append(validation_accuracy)

        else:
             print('training_accuracy => %.4f for step
%d'%(train_accuracy, i))
        train_accuracies.append(train_accuracy)
        x_range.append(i)
        # increase display_step
        if i%(display_step*10) == 0 and i and display_step<100:
            display_step *= 10
    # train on batch
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys, keep_prob:
DROPOUT})
```

Step 5: This step involves analyzing the results and visualizing it for the purpose of performance metrics.

First of all, we import the necessary libraries.

```
import seaborn as sns
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

import itertools
```

The following function checks the accuracy on validation set.

```
# check final accuracy on validation set
if(VALIDATION_SIZE):
    validation_accuracy = accuracy.eval(feed_dict={x: validation_images,
                                                   y_: validation_labels,
                                                   keep_prob: 1.0})
    print('validation_accuracy => %.4f'%validation_accuracy)
    plt.plot(x_range, train_accuracies,'-b', label='Training')
    plt.plot(x_range, validation_accuracies,'-g', label='Validation')
    plt.legend(loc='lower right', frameon=False)
    plt.ylim(ymax = 1.0, ymin = 0.0)
    plt.ylabel('accuracy')
    plt.xlabel('step')

    plt.show()
```

Now we read the test data from csv file and extract the relevant information about data.

```
saver = tf.compat.v1.train.Saver(tf.compat.v1.all_variables())
saver.save(sess, 'my-model1', global_step=0)
# read test data from CSV file
test_data = data[data.Usage == "PublicTest"]
test_data.head()
len(test_data)
test_pixels_values = test_data.pixels.str.split(" ").tolist()
test_pixels_values = pd.DataFrame(test_pixels_values, dtype=int)
test_images = test_pixels_values.values
test_images = test_images.astype(np.float)
test_images = test_images - test_images.mean(axis=1).reshape(-1,1)
test_images = np.multiply(test_images,100.0/255.0)
test_images = np.divide(np.subtract(test_images,each_pixel_mean),
each_pixel_std)
```

We now predict the test set. Here, using batches is more resource efficient. We also get the confusion matrix of the same.

```
predicted_lables = np.zeros(test_images.shape[0])
for i in range(0,test_images.shape[0]//BATCH_SIZE):
    predicted_lables[i*BATCH_SIZE : (i+1)*BATCH_SIZE] =
predict.eval(feed_dict={x: test_images[i*BATCH_SIZE : (i+1)*BATCH_SIZE],

keep_prob: 1.0})
```

```
test_data.emotion.values
```

```
confusion_matrix(test_data.emotion.values, predicted_lables)
```

```
array([[178,   3,  43,  79,  89,  13,  62],
       [ 13,  17,   7,   3,  11,   0,   5],
       [ 47,   0, 158,  70, 104,  52,  65],
       [ 39,   0,  22, 702,  45,  18,  69],
       [ 81,   1,  43, 121, 266,  13, 128],
       [ 21,   1,  30,  40,  14, 286,  23],
       [ 39,   0,  40, 119, 106,  10, 293]])
```

Confusion Matrix is also known as error matrix and is used as a performance metric for classification based models where the true values are already known. The following function prints and plots the confusion matrix. Normalization can be applied by setting `normalize=True`.

```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    print(cm)
```

```
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, int(cm[i, j]*100)/100.0,
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```