Convolutional Neural Network for gesture recognition

Chris Venour, August 2017

Introduction

This document describes a project in which a Convolutional Neural Network (CNN) was built to recognize the seven hand gestures shown in figure 1. The gestures in the first six images of figure 1 represent digits zero to five and the classifier also recognizes when no gesture is being made in a video stream.

Figure 1. The seven gestures the CNN is trained to recognize.

The CNN described in this document is trained from scratch on a dataset of gesture images created by Salience. Usually Transfer Learning solutions should be used for Computer Vision tasks because they often perform better and require considerably less training data than a neural network trained from scratch. However, Transfer Learning solutions – in which a model that has already been trained on a large publically available dataset of images, and is then fine-tuned on your domain-specific data – tend to be very large.

The Computer Vision task you are working on might be quite modest – as the project described in this document is – and may require only a small CNN with a few convolutional and max pooling layers. Training a small CNN from scratch might, therefore, be a better option in this case because the resulting CNN used in production will require fewer computing resources than a Transfer Learning solution. For example a CNN often used in Transfer Learning solutions is a VGG16 CNN that has been pretrained on ImageNet. This CNN has 16 layers and 14,714,688 parameters whereas the CNN trained from scratch, and used for our gestures project, is 33 times smaller; it has only 438,759 parameters.

1 Creating training and validation data

Pictures of different gestures were taken using a laptop's built-in camera. Training data consists of 2000 images per gesture (14000 images in total) and validation data consists of 400 images per gesture (2800 images in total).

The classifier was trained on gestures performed by the right hand and so this hand alone is recognized by the classifier. When capturing images from the laptop camera, the hand was tilted at different angles and was made to appear in different parts of the image - for example in the corners of the frame rather than always in the center of the picture. Similarly, pictures were taken of the hand at various distances from the camera. Training the CNN on pictures of the gestures in different orientations and positions makes the classifier more robust and better able to make accurate predictions when presented with noisy real-world data.

2 Preprocessing the data

The training data was (1) normalized (2) converted to black and white images using the MOG algorithm and (3) augmented before it was used to train the CNN. Details about each of these preprocessing steps are provided below.

Normalizing the data

Pixel values of an image in the video stream were re-scaled from the range 0-255 to the range 0-1 by dividing all the pixel values by 255. This preprocessing step, which is called normalization, was performed because large values fed into a Neural Network can slow down or even disrupt its learning. There are multiple ways of re-scaling or normalizing data. For example, a value can be transformed into its zscore, which expresses how many standard deviations the value is away from a mean, but a thorough discussion of different normalization methods is outside the scope of this document.

Applying the MOG transformation

During testing, two problems emerged:

1. The classifier performed poorly when lighting conditions during testing differed from the lighting conditions that existed when the training and validation data were collected.

2. The classifier performed poorly when different backgrounds appeared in the images.

Converting the color images from the video stream to grayscale images did not resolve problem #1 because grayscale pictures taken during the day still varied significantly from grayscale pictures taken at night. In other words, lighting conditions still made a picture of a gesture taken during the day differ significantly from a picture of that same gesture taken at night.

However, the "Mixture of Gaussian" (MOG) algorithm, which is available in a Python library called OpenCV, solved both problems #1 and #2. When applied to the video stream of color images, the MOG algorithm distinguishes moving foreground from static background by doing the following:

- Finding areas that change in consecutive frames and marking these areas as foreground.
- Finding areas that don't change in consecutive frames and marking these areas as background.

Areas in an image that are identified as static background are blacked out and can therefore be ignored by the CNN. In this way, the MOG transformation allows the classifier to focus on just the hand in an image and not be distracted by irrelevant background data, thus solving problem #2 above. Note that even if a user keeps their hand very still in front of a camera, the hand still moves enough to be regarded by the MOG algorithm as moving foreground rather than static background.

The MOG algorithm also solves problem #1 because it turns the background to black and the foreground to white. The resulting black and white images of the gestures, which can be seen in figure 2, are considerably more invariant to different lighting conditions than images that are simply converted to grayscale.

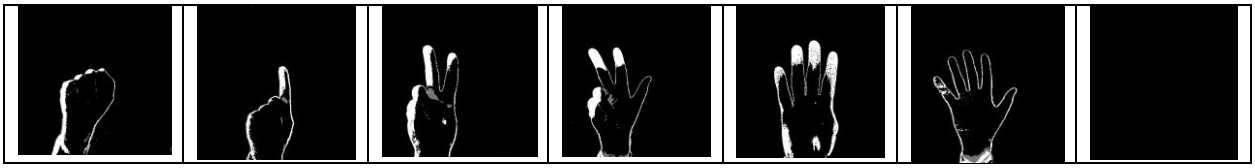

Figure 2. How images appear after MOG preprocessing

Augmenting the data using a Keras data generator

In this project, Keras data generators are used to load the training and validation data from disk and to perform augmentation operations on the training data before it is input into the CNN. Data augmentation generates more training data from existing training examples and this is useful because training a neural network on more data often makes it more accurate. Enlarging a training set via augmentation isn't as good as finding new data examples that aren't artificially created because the latter method adds entirely new information to the dataset whereas the data augmentation only makes slight transformations to existing information. But finding or creating new data can be very difficult and expensive, and performing data augmentation can be a good, albeit inferior option.

Random transformations of images can be performed to enlarge the amount of data - transformations such as zooming in or out of a picture, rotating the image by a few degrees, shifting an image horizontally and/or vertically, shearing it (i.e. distorting an object along an axis to simulate seeing it from a different angle) or changing the brightness of an image.

The following code implements a Keras data generator which normalizes images in the training set, shears them and zooms in and out of them by small random amounts:

```
#Images are rescaled, sheared and zoomed in and out
train_datagen = ImageDataGenerator(
    rescale=1. / 255,
    shear_range=0.2,
    zoom_range=0.2)
```

```
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(desired_width, desired_height),
    batch_size=batch_size,
    class_mode='categorical')
```

More specifically, the data generator implemented by this code performs the following sequence of steps:

1. Reads JPG image files on disk (that have already been transformed by the MOG algorithm).
2. Decodes the images files into 3D matrices of pixels.
3. Converts the matrices into TensorFlow tensors.
4. Rescales the matrices' pixel values from 0-255 to the 0-1 range.
5. Resizes the original images from 500 x 550 to 100 x 100. (This preprocessing step significantly decreases the time the CNN requires to process an image).
6. Augments the training data by creating new images in which the original images have been sheared by a random amount. The random amount varies for each image and ranges between 0 and 0.2 radians.
7. Augments the training data by creating new images in which the original images have been zoomed in or out by a random amount. The random amount varies for each image and ranges between 0 and 20%.
8. Generates labels for the images based on the name of the directory containing them, and groups the images into batches.

The validation data generator is constructed in a similar way, but it only rescales and resizes the images - it does not augment the validation data by shearing or zooming in and out of images:

```
#Images are only rescaled
test_datagen = ImageDataGenerator(rescale=1. / 255)

validation_generator = test_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(desired_width, desired_height),
    batch_size=batch_size,
    class_mode='categorical')
```

3 Building the CNN classifier

The CNN consists of three convolution layers, each of which is followed by a max-pooling layer, and two fully-connected layers appear at the end of the network. The final layer of the model consists of seven nodes and a sigmoid activation because the classifier's goal is to detect seven different gestures. Node 1 in that final layer computes a probability that the image input into the classifier contains gesture #1, node 2 computes a probability that the gesture is gesture #2 etc.

Only eleven lines of Keras code are required to build this CNN:

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation = 'relu', input_shape = (desired_width,
desired_height, 3)))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64, activation = 'relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes, activation='softmax'))
```

Keras has a command called `model.summary()` which displays the architecture of the CNN and the output from this command can be seen in figure 6.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 98, 98, 32)        896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 49, 49, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 47, 47, 32)        9248
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 23, 23, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 21, 21, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 10, 10, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 6400)              0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                409664
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 7)                 455
=================================================================
```

Figure 3. The height and width of the image appear in the rectangle. Note how the height and width of the image decrease after a convolution or max-pooling operation because no padding is applied (i.e. padding='valid').

Images fed into the CNN have an initial height and width of 100 x 100 pixels and we can see in the first row of the boxed portion of figure 3 that, after the first convolution, the image has been reduced to 98 x 98 pixels. The boxed portion of figure 3 shows that as the image passes through more convolutional and max-pooling layers, its height and width continue to decrease[1].

[1] The formula for determining the height and width of the image as it passes through `Conv2D` and `MaxPooling2D` layers is $floor(\frac{n+2p-f}{s}+1)$ where n is the original height and width of the image, p is the amount of padding, f is the filter size and s is the size of the stride. The default setting for a `Conv2D` layer in Keras is padding='valid' which means no padding is performed on the image (p = 0) and the default stride is s=1. So the 100 x 100 image in our example becomes a 98 x 98 image after passing through the first Convolutional layer because $floor(\frac{100+2*0-3}{1}+1)=98$. The default setting for a `MaxPooling2D` layer is p=0 and the stride defaults to the `pool_size` of the pooling filter which is 2 in our code. So the 98 x 98 image becomes a 49 x 49 image because $floor(\frac{98+2*0-2}{2}+1)=49$.