Notes de cours CADL - session-1

cours Kadenze - session-1

Thibaut Marmey

October 24, 2018

- Learn the basic idea behind machine learning: learning from data and discovering representations
- Learn how to preprocess a dataset using its mean and standard deviation
- Learn the basic components of a Tensorflow Graph

Contents

1 Introduction

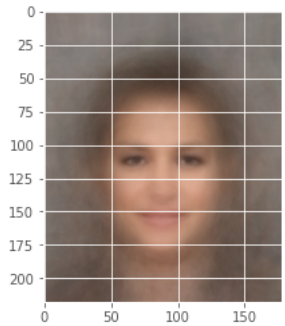
1.1 Généralités

- Deep-learning in a type of Machine Learning
- *Deep* because it is composed of many layers of *Neural Networks*
- Other valuable branches of Machine Learning :
  - Rinforcement Learning
  - Dictionary Learning
  - Probabilistic Graphical Models and Bayesian Methods (Bishop)
  - Genetic and Evolutionary Algorithms
- The differents ways an object can appear in an image is called *invariance*
- The dataset teaches the algorithm how to see the world, but only the world of this dataset
- Existing data :

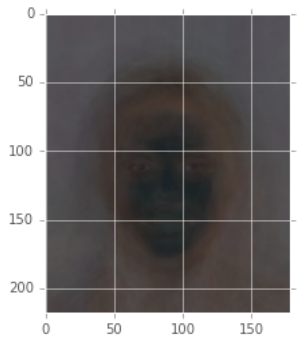
- MNIST
- CalTech
- CelebNet
- ImageNet
- LFW
- CIFAR10, CIFRA100, MS Coco...

1.2 Preprocessing Data

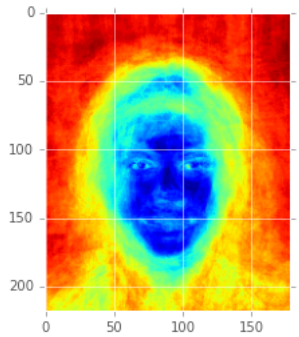
- Collect the images into a batch configuration. With this configuration, it's easier to make some computation over all the data.
  This means, the data is in a single *numpy* variable : *data = np.array(imgs)*
- Compute the Mean and Deviation of Images (of the batch channel)
  - ```
    mean_img = np.mean(data, axis=0) #mean of each col
    plt.imshow(mean_img.astype(np.uint8))
    ```

    This describes what most the dataset looks like.
  - ```
    std_img = np.std(data, axis=0)
    plt.imshow(std_img.astype(np.uint8))
    ```

    This describes where the changes are the most likely to appear in the dataset of images.
  - ```
    plt.imshow(np.mean(std_img, axis=2).astype(np.uint8))
    ```
    ```



This describes how every color channel will vary as a heatmap.

* Red part : not the best representation of the image
* Blue part : the less likely that our mean image is far off from any other possible image

1.3 Dataset preprocessing

- We are trying to build a model that understands invariances (different of vision of an object, localization in the image, etc...)
- If we use DL to learn something complex in the data, it starts by modeling both the mean and standard deviation or our dataset.
- Speed up by "preprocessing" the dataset by removing the mean and standard deviation : it's called *normalization*.
  Subsctracting the mean and dividing by the standard deviation.
- Look at the dataset with another way : array into a 1 dimensional array.

```
flattened = data.ravel()
```

- Visualize the **"distribution"**, or range and frequence of possible values. This tell us if **the data is predictable or not**.
  plt.hist(data.ravel(), n takes the min and max values of the *data* array, and divide this interval in n subintervals.

```
plt.hist(flattened.ravel(), 255) #values are grouping in 255 bins
```

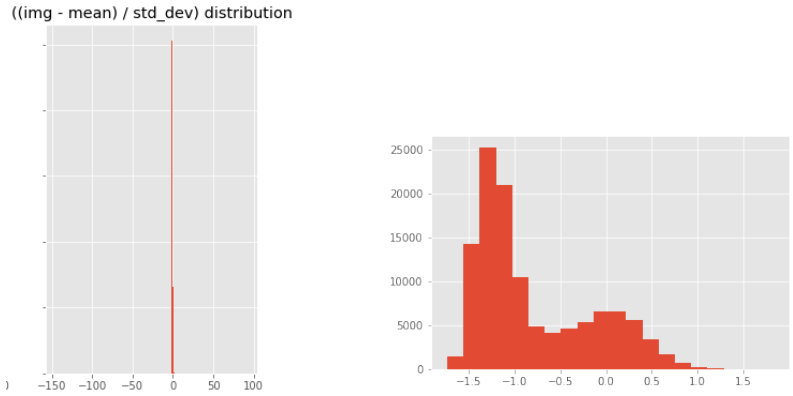
  It tells us if something seems to happen more than anything else. If it does, the neural network will take advantage of that.

- Normalization :

```
plt.hist(((data[0] - mean_img) / std_img).ravel(), bins)
```

  The data has been squished into a peak. Change the scale of the hist.
  The data is concentrated between two values. The effect of normalizing : most of the data will be around 0, where some deviations of it will follow between the two

values.



(a) (b)

- If the normalization doesn't look like this :
  - get more data to calculate our mean/std deviation
  - try another method of normalization
  - not bother with normalization at all
- Other options of normalization :
  - local contrast normalization for images
  - PCA based normalization

2 Tensorflow Basics

2.1 Basics

- Working with Google's Library for Numerical Computation, TensorFlow
- Different approach for doing the things above.
- Instead of computing things immediately, we first define things that we want to compute later using what's called a *Graph*.
- Import the tensorflow library

```
import tensorflow as tf
```

- Range of numbers
  - with *numpy* :

```
np.linspace(-n, n, nbSubIntervals) #return list of 100 float64
```

  - with *Tensorflow* :

```
x = tf.linspace(-n,n,nbSubIntervals)
#return Tensor("LinSpace:0", shape=(100,), dtype=float32)
```

    - *LinSpace* : name
    - *shape* : dimension and the number of values
    - *dtype* : type of the values

- *tf.Tensor* and *numpy.array* return different type of values.
  No values printed because it actually hasn't computed its values yet. It just refers to the output of a *tf.Operation*, already added to TF's default computational **graph**. The result is the returned **Tensor object**.

- Inspect "default" graph where all the operation have been added :

```
tf.get_default_graph()
```

  - Get the list of all added operation.

```
[op.name for op in g.get_operations]
```

- The result of a *tf.Operation* is a *tf.Tensor*

- Create a *tf.Session* to actually compute anything. It is responsible for evaluating the *tf.Graph*.

```
sess = tf.Session() #create session
#compute anything created in the tensorflow graph
commputed_x = sees.run()
sess.close()  #close session
```

- In iPython's interactive console, create an *tf.InteractiveSession*

```
sess = tf.InteractiveSession()
#The session is open for the rest of the lecture in Jupyter
x.eval()
```

- Access to the shape : *x.get_shape()*

- Create Gaussian curve (also refered by *bell curve* or *normal curve*. It should resemble a normalized histogram. It needs two variables *the mean value* (the curve is centred on it) and *the standard deviation*.

```
mean = 0.0
sigma = 1.0
z = (tf.exp(tf.negative(tf.pow(x - mean, 2.0) / (2.0 * tf.pow(sigma,
   2.0)))) * (1.0 / (sigma * tf.sqrt(2.0 * 3.1415))))
```

  Nothing has been computed. Just added operations to TF's graph. If we want the value or output, we have to ask the part of the graph we are interested.

- Interactive session is already created, call the *eval()* function on the name of the interested Tensor.

2.2 Convolution

- Creating a 2-D Gaussian Kernel. Can be done by multiplying a vector by its transpose.
  tf.reshape(tensor, list)

```
# Let's store the number of values in our Gaussian curve.
ksize = z.get_shape().as_list()[0]
# Let's multiply the two to get a 2d Gaussian
z_2d = tf.matmul(tf.reshape(z, [ksize, 1]), tf.reshape(z, [1, ksize]))
# Execute the graph
plt.imshow(z_2d.eval())
```

- Common operation in DL : **convolution**
  A way of filtering information. Here a link to visualize gaussian filters.

- Import a RGB image and convert it in a grayscale image with *scikit-image* library.

```
from skimage import color
from skimage import io
img = color.rgb2gray(io.imread('file')).astype(np.float32)
# Think to convert in np.float32 !
plt.imshow(img, cmap='gray')
```

  The shape of *img* is 2D. For image convultion with TF we need the batch dimension (N*H*W*C). With one grayscale image the shape is 1*H*W*1 (1 image, 1 channel)
  .
  Let's use the TF reshape function :

```
img_4d = tf.reshape(img, [1, img.shape[0], img.shape[1], 1])
#Tensor("Reshape_50:0", shape=(1, 1920, 1920, 1), dtype=float32)
# float32 ok !
```

- Reshape Gaussian Kernel to 4d (batch dimension) but the kernel dimension is different :
  *Kernel Height * K_W * Number of Input Channels * Num of Output Channels*

```
z_4d = tf.reshape(z_2d, [ksize, ksize, 1, 1])
```

- Convolve image with Gaussian Kernel

```
convolfed = tf.nn.conv2D(img_4D, strides=[1,1,1,1], pading='SAME')
```

  - *strides* : how to move our kernel across the image. Basically, two sets of parameters :
    * [1, 1, 1, 1], which means, we are going to convolve every single image, every pixel, and every color channel by whatever the kernel is.
    * [1, 2, 2, 1], which means, we are going to convolve every single image, but every other pixel, in every single color channel.

  - *padding* : what to do at the borders :
    - *'SAME'*, same dimensions as the original image (same dimensions going in and going out)
    - *'VALID'*, the dimensions going out will change

- Stopped at **Modulating the Gaussian with a Sine Wave to create Gabor Kernel**