Notes de cours CADL - session-1

cours Kadenze - session-1

Thibaut Marmey

October 26, 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)

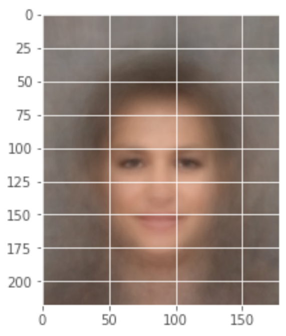
  - Genetic and Evolutionary Algorithms

- 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

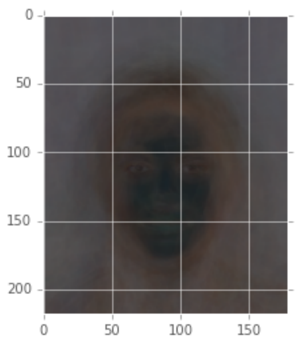
- Collect the images into a batch configuration. With this configuration, it's easier to make some computation over all the data.
  This means, the data is in a single *numpy* variable : *data = np.array(imgs)*
- Compute the Mean and Deviation of Images (of the batch channel)

  -
    ```
    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.

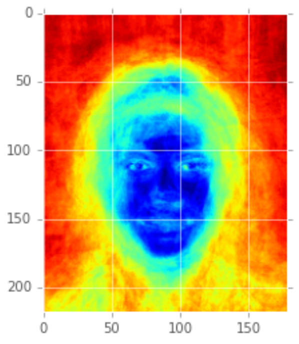
  -
    ```
    std_img = np.std(data, axis=0)
    plt.imshow(std_img.astype(np.uint8))
    ```

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()
  ```

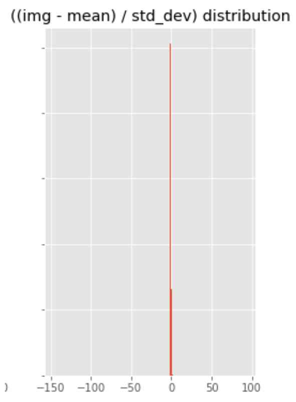
- Visualize the **"distribution"**, or range and frequence of possible values. This tell us if **the data is predictable or not**.
  plt.hist(data.ravel(), n takes the min and max values of the *data* array, and divide this interval in *n* subintervals.

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

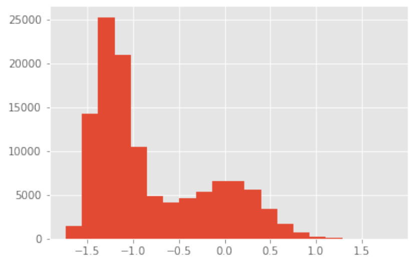
  It tells us if something seems to happen more than anything else. If it does, the neural network will take advantage of that.
- Normalization :

  ```
  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))))
```

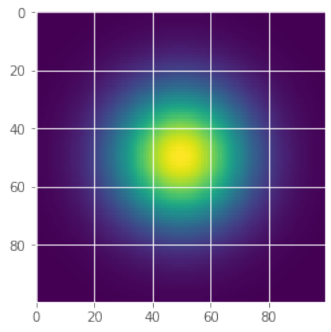
  Nothing has been computed. Just added operations to TF's graph. If we want the value or output, we have to ask the part of the graph we are interested.

- 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

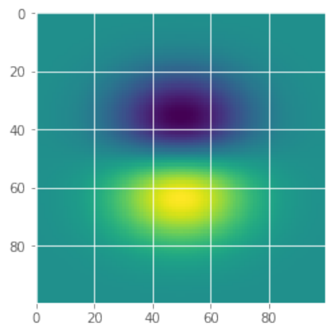
- Modulating the Gaussian with a Sine wave to create Gabor Kernel
  Convolution kernel called a Gabor : like the Gaussian Kernel but using a sine wave to modulate.

- Create an interval from -n to +n standard deviations with nb subintervals and the sin wave.

```
xs = tf.linspace(-n, +n, nb)
ys = tf.sin(xs)
ys = tf.reshape(ys, [ksize, 1])
ones = tf.ones((1, ksize))
wave = tf.matmul(ys, ones)
```

- Create the Gabor Kernel with the sin wave created before

```
gabor = tf.multiply(wave, z_2d) #z_2d 2d-Gaussian
# tf.multiply() does element-wise multiplication
# tf.matmul() does matrix multiplication
```



- Placeholders *tf.placeholder(type, shape= , name=)* : specifying in our graph which elements we wanted to be specified later. Not sure what these are yet but we know they will fit in the graph (generally the input and output of the network).

- Rewrite convolution using placeholder for the image and the kernel and setting image dimensions to *None*None* (special for placeholders).

```
# This is a placeholder which will become part of the tensorflow graph,
   but which we have to later explicitly define whenever we run/evaluate
   the graph.
img = tf.placeholder(tf.float32, shape=[None, None], name='img')
```

- Reshape 2d image to a 3d tensor and again to batch dimension using *tf.expand_dims(img, axis)*.
  At the axis the channel dimension of 1 is added.

```
# before dim : H * W
img_3d = tf.expand_dims(img, 2)
# img_3d.get_shape() = H * W * 1
img_4d = tf.expand_dims(img_4d, 0)
# img_4d.get_shape() = 1 * H * W * 1
```

- Create another set of placeholders for Gabor's parameters

```
mean = tf.placeholder(tf.float32, name='mean')
sigma = tf.placeholder(tf.float32, name='sigma')
ksize = tf.placeholder(tf.float32, name='ksize')
```

- Redo the commands to build 2d gaussian kernel, sin wave, and finally *gabor_4d* :

```
gabord_4d = tf.reshape(gabor, [ksize, ksize, 1,1])
```

- At the end, do the convolution :

```
convolved = tf.nn.conv2d(img_4d, gabor_4d, strides=[1, 1, 1, 1],
   padding='SAME', name='convolved')
convolved_img = convolved[0, :, :, 0]
```

- At this point, not able to compute the convolution *convolved_img.eval()* because we have to specify all the placeholders required for the computation.
  Get the error : "feed value for placeholder tensor".

- Feed a value :

```
res = convolved_img.eval(feed_dict={
     img: data.camera(),
     mean:0.0, sigma:1.0,
     ksize:100
     })
```

3 Homework

3.1 Create a small dataset

- Use extension of chrome web store : link
- Dataset download : images with cats
- Load the dataset and create a 4d array (batch dimension):

```
# load image from directory
filenames = [os.path.join('dirname', fname) for fname in
    os.listdir(dirname)]
# In filenames, we have the link of each pictures
# Read every filename as an RGB image
imgs = [plt.imread(fname)[:,:,:3] for fname in filenames]
# Crop the picture to square and the same size (size, size)
------
# Create the 4-D array like (number of images * H * W * color)
# And convert in np.float32
imgs = np.array(imgs).astype(np.float32)
```

- Open the session for Jupyter Notebook

```
sees = tf.InteractvieSession()
```

3.2 Compute the mean

- Calculate the mean with *tf.reduce_mean(input_tensor, axis, ...)* to get th emean color image (size,size,3). Get the mean of every pixel.

```
mean_img_op = tf.mean_reduse(imgs, axis=0)
```

- Run operation in the session

```
mean_img = sess.run(mean_img_op)
```

- Create *assert* debuging error to be sure of what we have as input or output like the size of the image we are working on

```
assert(mean_img_op == (size,size,3))
```

if plotting the *mean_img* we have :

  - something "regular" and "predictable" means the dataset is good
  - otherwise gray blob, not so much to see : not a lot in common un the dataset, or need to work deeper to find something else, or the dataset is too small to have something relevant.

3.3 Compute the Standard Deviation

- Compute the standard deviation on each color channel. This means to keep the 4d dimensions of the *imgs* data.

  std formula : $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$

  - N : number of image
  - x_i : image i
  - μ : mean of the images of the dataset

```
# Calcul of mu
mean_img_4d = tf.reduce_mean(imgs, axis=0, keepdims=True)
# keep_dims = True ask the output to be to the same dimension of the
    input (so 3 color channels)
# Do the substraction x_i - mu
substraction = img - mean_img_4d
# Application of the formula
std_img_op = tf.sqrt(tf.reduce_mean((substraction * substraction),
    axis=0))
# Computation of the std using the session
std_img = sess.run(std_img_op)
```

3.4 Normalize the Dataset

- Normalization formula : *(imgs - mean_img)/std_img*
- Normalization is to center the data and make sure the range is similar in order to get stable gradients.
- Batch normalization : subtraction per channel uses to center the data around zero mean for each channel (R,G,B). Helps network to learn faster since gradients act uniformly for each channel.

3.5 Convolve the Dataset

- Build a "hand-crafted" feature detector : Gabor Kernel. The shape of a 4d kernel is : *(K_H, K_W, Channel input, Channel output)*. With this tutorial we can use :

```
ksize = 32
# Channel input = 3 (R, G, B)
kernel = np.concatenate([utils.gabor(ksize)[:,:,np.newaxis] for i in
    range(3)], axis=2)
kernel_4d = np.reshape(kernel, [ksize, ksize, 3, 1])
convolved = utils.convolve(imgs, kernel_4d)
# function from tutorial using tf.nn.conv2d(img, kernel, strides,
    padding)
```

3.6 Sort the Dataset

- Use the mean value of each convolved image's output for sorting.
  - Calculate the sum value : *tf.reduce_sum* of each image
    Calculate the mean value : *tf.reduce_mean* of each image

    ```
    # From 100 3d images (100*100*1C) we want to have 100 1d vectors
    flattened = tf.reshape(convolved, [100, 100*100])
    # 100 sum results of the 1d vetors
    values = tf.reduce_sum(flattened, axis=1)
    ```

  - Sort the images by the sum values with *tf.nn.top_k* (returns the values [0] and the indexes [1]).

    ```
    idxs_op = tf.nn.top_k(values, k=100)[1]
    idxs = sess.run(idxs_op)
    # without np.array sorted_imgs is a list and we want numpy.ndarray
       type
    sorted_imgs = np.array(imgs[idx_i] for idx_i in idxs])
    ```

- What we have : "low level" eges essentially in our case are not very good at describing the really interesting sapects of the dataset.