☰

Preprocessing for deep learning: from covariance matrix to image whitening

27-08-2018 | HADRIENJ Follow @_hadrienj | COMPUTER-VISION PYTHON NUMPY DEEP-LEARNING

Last update: Jan. 2021

A notebook version of this post can be found here on Github.

The goal of this post/notebook is to go from the basics of data preprocessing to modern techniques used in deep learning. My point is that we can use code (Python/Numpy etc.) to better understand abstract mathematical notions! Thinking by coding! 💥

We will start with basic but very useful concepts in data science and machine learning/deep learning like variance and covariance matrix and we will go further to some preprocessing techniques used to feed images into neural networks. We will try to get more concrete insights using code to actually see what each equation is doing!

We call preprocessing all transformations on the raw data before it is fed to the machine learning or deep learning algorithm. For instance, training a convolutional neural network on raw images will probably lead to bad classification performances (Pal & Sudeep, 2016). The preprocessing is also important to speed up training (for instance, centering and scaling techniques, see Lecun et al., 2012; see 4.3).

Here is the syllabus of this tutorial:

1. **Background**: In the first part, we will get some reminders about variance and covariance and see how to generate and plot fake data to get a better understanding of these concepts.

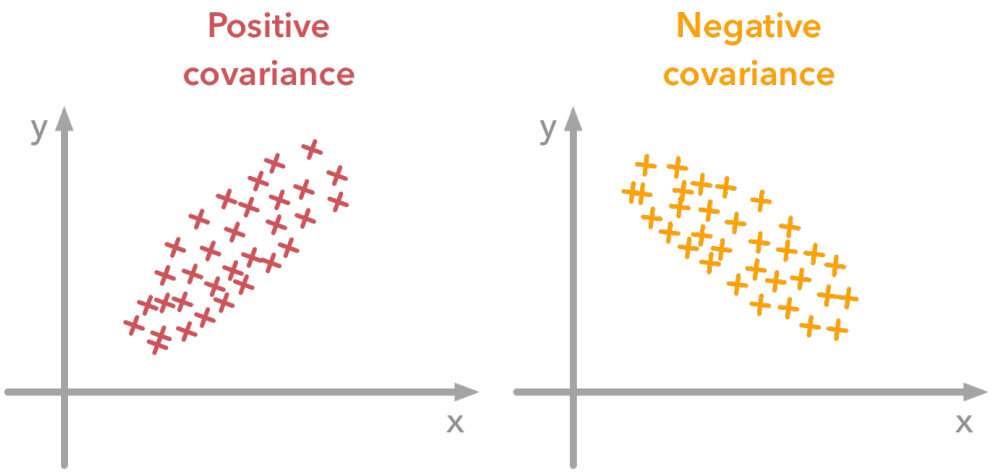
3. **Whitening images**: In the third part, we will use the tools and concepts gained in 1. and 2. to do a special kind of whitening called Zero Component Analysis (ZCA). It can be used to preprocess images for deep learning. This part will be very practical and fun ⛄!

Feel free to fork the notebook. For instance, check the shapes of the matrices each time you have a doubt :)

1. Background

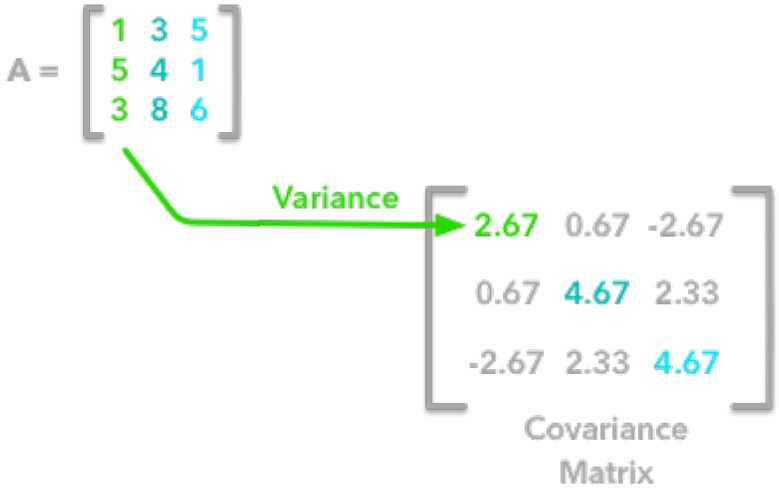
A. Variance and covariance

The variance of a variable describes how much the values are spread. The covariance is a measure that tells the amount of dependency between two variables. A positive covariance means that values of the first variable are large when values of the second variables are also large. A negative covariance means the opposite: large values from one variable are associated with small values of the other. The covariance value depends on the scale of the variable so it is hard to analyse it. It is possible to use the correlation coefficient that is easier to interpret. It is just the covariance normalized.



A positive covariance means that large values of one variable are associated with big values from the other (left). A negative covariance means that large values of one variable are associated with small values of the other one (right).

variance of each vector.



A matrix A and its matrix of covariance. The diagonal corresponds to the variance of each column vector.

Let's just check with the formula of the variance:

$V(\boldsymbol{X}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$

with n the length of the vector, and $\bar{x}$ the mean of the vector. For instance, the variance of the first column vector of $\boldsymbol{A}$ is:

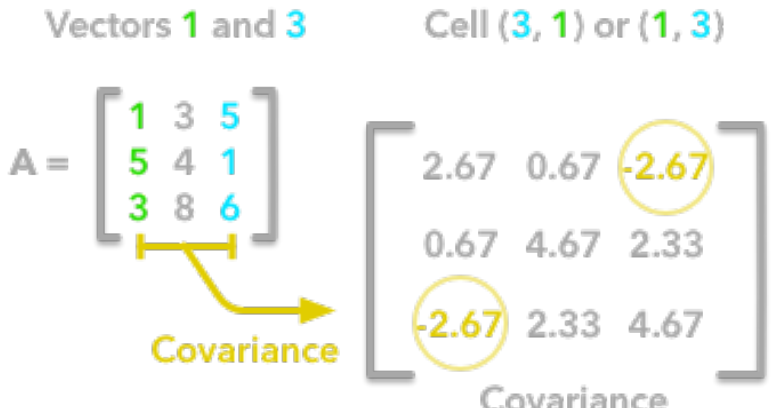
$V(\boldsymbol{A}_{:,1}) = \frac{(1-3)^2+(5-3)^2+(3-3)^2}{3} = 2.67$

This is the first cell of our covariance matrix. The second element on the diagonal corresponds of the variance of the second column vector from $\boldsymbol{A}$ and so on.

Note: the vectors extracted from the matrix $\boldsymbol{A}$ correspond to the columns of $\boldsymbol{A}$.

Covariance

The other cells correspond to the covariance between two column vectors from $\boldsymbol{A}$. For instance, the covariance between the first and the third column is located in the covariance matrix as the column 1 and the row 3 (or the column 3 and the row 1).



☰

The position in the covariance matrix. Column corresponds to the first variable and row to the second (or the opposite). The covariance between the first and the third column vector of A is the element in column 1 and row 3 (or the opposite = same value).

Let's check that the covariance between the first and the third column vector of $\boldsymbol{A}$ is equal to -2.67. The formula of the covariance between two variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ is:

$$cov(\boldsymbol{X}, \boldsymbol{Y}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

The variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ are the first and the third column vectors in the last example. Let's split this formula to be sure that it is crystal clear:

1. $(x_1 - \bar{x})$. The sum symbol means that we will iterate on the elements of the vectors. We will start with the first element ($i = 1$) and calculate the first element of $\boldsymbol{X}$ minus the mean of the vector $\boldsymbol{X}$.
2. $(x_1 - \bar{x})(y_1 - \bar{y})$. Multiply the result with the first element of $\boldsymbol{Y}$ minus the mean of the vector $\boldsymbol{Y}$.
3. $\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$. Reiterate the process for each element of the vectors and calculate the sum of all results.
4. $\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$. Divide by the number of elements in the vector.

Example 1.

Let's start with the matrix $\boldsymbol{A}$:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 3 & 5 \\ 5 & 4 & 1 \\ 3 & 8 & 6 \end{bmatrix}$$

We will calculate the covariance between the first and the third column vectors:

$$\boldsymbol{X} = \begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix}$$

and

$$\boldsymbol{Y} = \begin{bmatrix} 5 \\ 1 \\ 6 \end{bmatrix}$$

$\bar{\boldsymbol{x}} = 3$, $\bar{\boldsymbol{y}} = 4$ and $n = 3$ so we have:

$$cov(X, Y) = \frac{(1-3)(5-4)+(5-3)(1-4)+(3-3)(6-4)}{3} = \frac{-8}{3} = -2.67$$

Ok, great! That the value of the covariance matrix.

Now the easy way! With Numpy, the covariance matrix can be calculated with the function `np.cov`.

Let's create the array first:

```
A = np.array([[1, 3, 5], [5, 4, 1], [3, 8, 6]])
A
```

```
array([[1, 3, 5],
       [5, 4, 1],
       [3, 8, 6]])
```

Now we will calculate the covariance with the Numpy function:

```
np.cov(A, rowvar=False, bias=True)
```

```
array([[ 2.66666667,  0.66666667, -2.66666667],
       [ 0.66666667,  4.66666667,  2.33333333],
       [-2.66666667,  2.33333333,  4.66666667]])
```

Looks good!

Finding the covariance matrix with the dot product

There is another way to compute the covariance matrix of $\boldsymbol{A}$. You can center $\boldsymbol{A}$ around 0 (subtract the mean of the vector to each element of the vector to have a vector of mean equal to 0, *cf.* below), multiply it with its own transpose and divide by the number of observations. Let's start with an implementation and then we'll try to understand the link with the previous equation:

```
def calculateCovariance(X):
    meanX = np.mean(X, axis = 0)
    lenX = X.shape[0]
    X = X - meanX
    covariance = X.T.dot(X)/lenX
    return covariance
```

Let's test it on our matrix $\boldsymbol{A}$:

```
calculateCovariance(A)
```

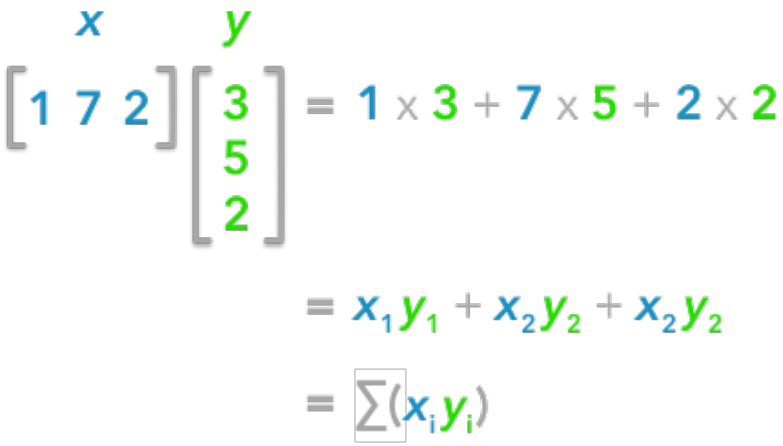
```
array([[ 2.66666667,  0.66666667, -2.66666667],
       [ 0.66666667,  4.66666667,  2.33333333],
       [-2.66666667,  2.33333333,  4.66666667]])
```

We end up with the same result as before!

☰

$\boldsymbol{X}^T\boldsymbol{Y} = \sum_{i=1}(x_i)(y_i)$

That's right, it is the sum of the products of each element of the vectors:



The dot product corresponds to the sum of the products of each element of the vectors.

If n is the number of elements in our vectors and that we divide by n:

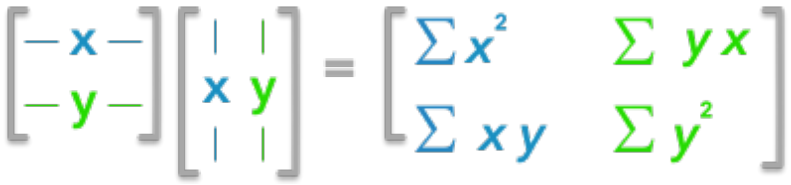
$\frac{1}{n}\boldsymbol{X}^T\boldsymbol{Y} = \frac{1}{n}\sum_{i=1}^{n}(x_i)(y_i)$

You can note that this is not too far from the formula of the covariance we have seen above:

$cov(\boldsymbol{X}, \boldsymbol{Y}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$

The only difference is that in the covariance formula we subtract the mean of a vector to each of its elements. This is why we need to center the data before doing the dot product.

Now if we have a matrix $\boldsymbol{A}$, the dot product between $\boldsymbol{A}$ and its transpose will give you a new matrix:



If you start with a zero-centered matrix, the dot product between this matrix and its transpose will give you the variance of each vector and covariance between them, that is to say the covariance matrix.

This is the covariance matrix! 🌵

B. Visualize data and covariance matrices

In order to get more insights about the covariance matrix and how it can be useful, we will create a function used to visualize it along with 2D data. You will be able to see the link between the covariance matrix and the data.

☰

used to create gradients of color. Small values will be colored in light green and large values in dark blue. The data is represented as a scatterplot. We choose one of our palette colors, but you may prefer other colors 🌈.

```
def plotDataAndCov(data):
    ACov = np.cov(data, rowvar=False, bias=True)
    print 'Covariance matrix:\n', ACov

    fig, ax = plt.subplots(nrows=1, ncols=2)
    fig.set_size_inches(10, 10)

    ax0 = plt.subplot(2, 2, 1)

    # Choosing the colors
    cmap = sns.color_palette("GnBu", 10)
    sns.heatmap(ACov, cmap=cmap, vmin=0)

    ax1 = plt.subplot(2, 2, 2)

    # data can include the colors
    if data.shape[1]==3:
        c=data[:,2]
    else:
        c="#0A98BE"
    ax1.scatter(data[:,0], data[:,1], c=c, s=40)

    # Remove the top and right axes from the data plot
    ax1.spines['right'].set_visible(False)
    ax1.spines['top'].set_visible(False)
```

C. Simulating data

Uncorrelated data

Now that we have the plot function, we will generate some random data to visualize what the covariance matrice can tell us. We will start with some data drawn from a normal distribution with the Numpy function `np.random.normal()`.

Drawing sample from a normal distribution with Numpy.

This function needs the mean, the standard deviation and the number of observations of the distribution as input. We will create two random variables of 300 observations with a standard deviation of 1. The first will have a mean of 1 and the second a mean of 2. If we draw two times 300 observations from a normal distribution, both vectors will be uncorrelated.

```
np.random.seed(1234)
a1 = np.random.normal(2, 1, 300)
a2 = np.random.normal(1, 1, 300)
A = np.array([a1, a2]).T
A.shape
```

☰

Note 1: We transpose the data with `.T` because the original shape is (2, 300) and we want the number of observations as rows (so with shape (300, 2)).

Note 2: We use `np.random.seed` function for reproducibility. The same random number will be used the next time we run the cell!

Let's check how the data looks like:

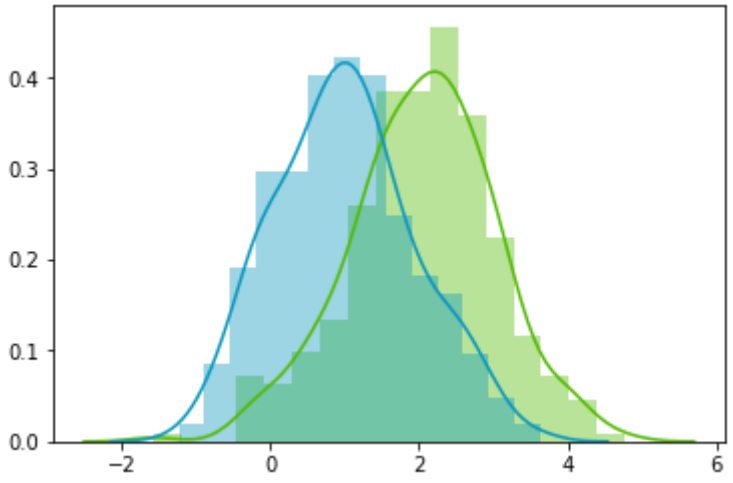
```
A[:10,:]
```

```
array([[ 2.47143516,  1.52704645],
       [ 0.80902431,  1.7111124 ],
       [ 3.43270697,  0.78245452],
       [ 1.6873481 ,  3.63779121],
       [ 1.27941127, -0.74213763],
       [ 2.88716294,  0.90556519],
       [ 2.85958841,  2.43118375],
       [ 1.3634765 ,  1.59275845],
       [ 2.01569637,  1.1702969 ],
       [-0.24268495, -0.75170595]])
```

Nice, we have our two columns vectors.

Now, we can check that the distributions are normal:

```
sns.distplot(A[:,0], color="#53BB04")
sns.distplot(A[:,1], color="#0A98BE")
plt.show()
plt.close()
```

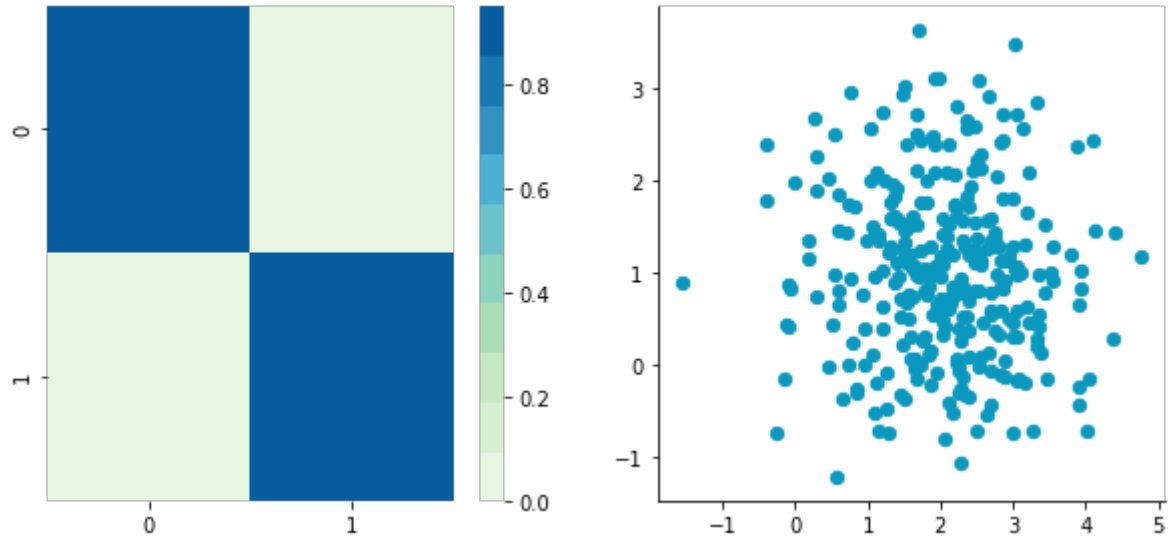


Looks good! We can see that the distributions have equivalent standard deviations but different means (1 and 2). So that's exactly what we have asked for!

Now we can plot our dataset and its covariance matrix with our function:

```
plt.close()
```

```
Covariance matrix:
[[ 0.95171641 -0.0447816 ]
 [-0.0447816   0.87959853]]
```



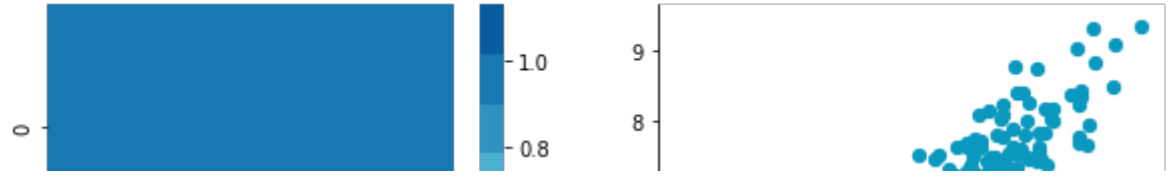
We can see on the scatterplot that the two dimensions are uncorrelated. Note that we have one dimension with a mean of 1 and the other with the mean of 2. Also, the covariance matrix shows that the variance of each variable is very large (around 1) and the covariance of columns 1 and 2 is very small (around 0). Since we insured that the two vectors are independent this is coherent (the opposite is not necessarily true: a covariance of 0 doesn't guaranty independency (see here).

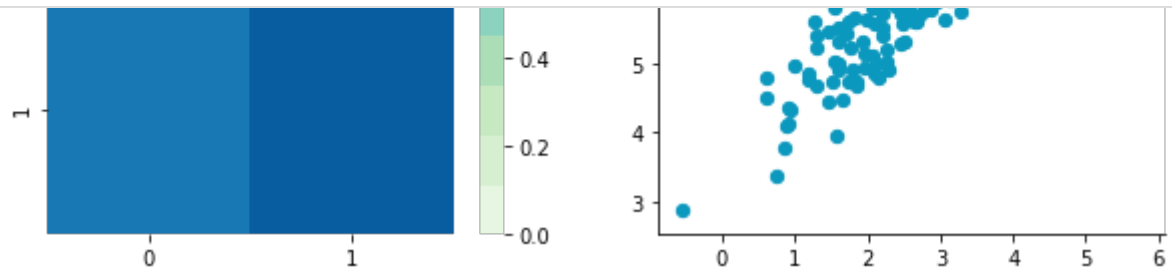
Correlated data

Now, let's construct dependent data by specifying one column from the other one.

```
np.random.seed(1234)
b1 =  np.random.normal(3, 1, 300)
b2 = b1 + np.random.normal(7, 1, 300)/2.
B = np.array([b1, b2]).T
plotDataAndCov(B)
plt.show()
plt.close()
```

```
Covariance matrix:
[[ 0.95171641  0.92932561]
 [ 0.92932561  1.12683445]]
```





The correlation between the two dimensions is visible on the scatter plot. We can see that a line could be drawn and used to predict $\boldsymbol{y}$ from $\boldsymbol{x}$ and vice versa. The covariance matrix is not diagonal (there are non-zero cells outside of the diagonal). That means that the covariance between dimensions is non-zero.

That's great! ⚡ We now have all the tools to see different preprocessing techniques.

2. Preprocessing

A. Mean normalization

Mean normalization is just removing the mean from each observation.

$$\boldsymbol{X}' = \boldsymbol{X} - \bar{x}$$

where $\boldsymbol{X}'$ is the normalized dataset, $\boldsymbol{X}$ the original dataset and $\bar{x}$ the mean of $\boldsymbol{X}$.

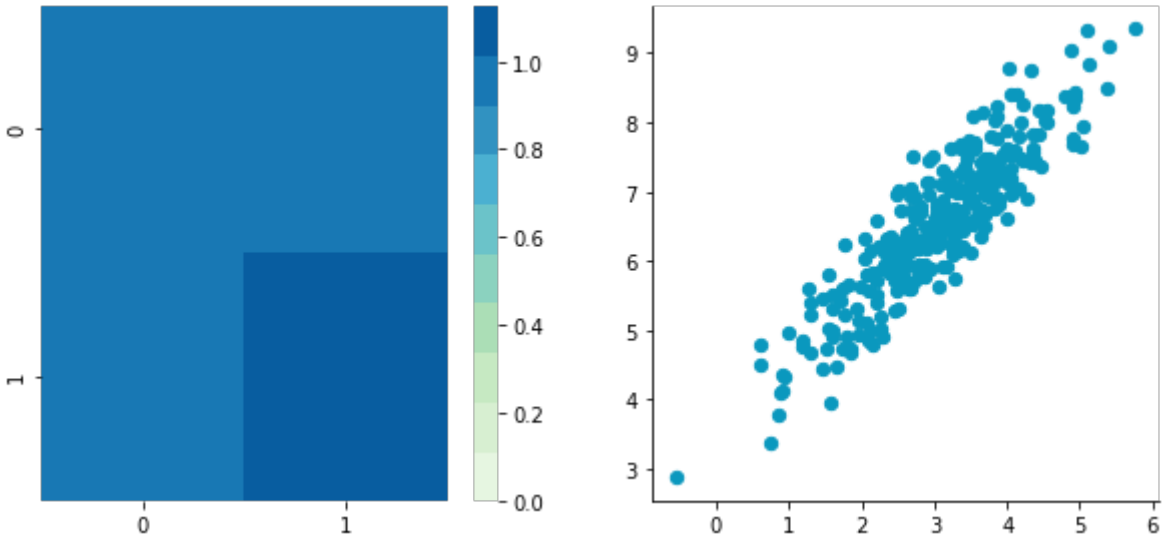
It will have the effect of centering the data around 0. We will create the function `center()` to do that:

```
def center(X):
    newX = X - np.mean(X, axis = 0)
    return newX
```

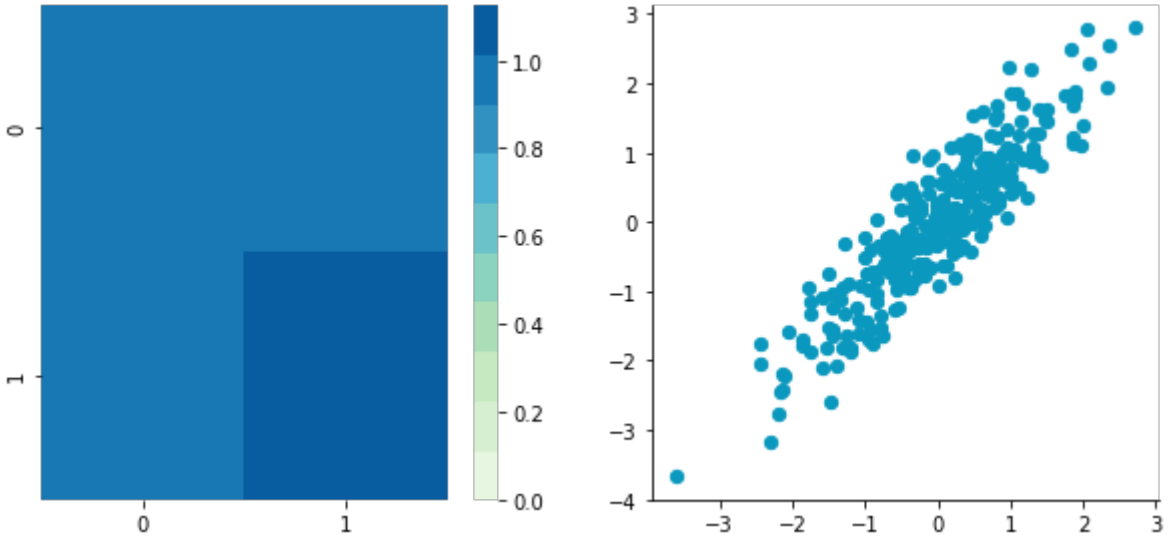
Let's give it a try with the matrix $\boldsymbol{B}$ we have created above:

```
BCentered = center(B)

print 'Before:\n\n'

plotDataAndCov(B)
plt.show()
plt.close()

print 'After:\n\n'

plotDataAndCov(BCentered)
plt.show()
plt.close()
```

```
Covariance matrix:
[[ 0.95171641  0.92932561]
 [ 0.92932561  1.12683445]]
```




```
After:


Covariance matrix:
[[ 0.95171641  0.92932561]
 [ 0.92932561  1.12683445]]
```



The first plot shows again the original data $\boldsymbol{B}$ and the second plot shows the centered data (look at the scale).

B. Standardization

The standardization is used to put all features on the same scale. The way to do it is to divide each zero-centered dimension by its standard deviation.

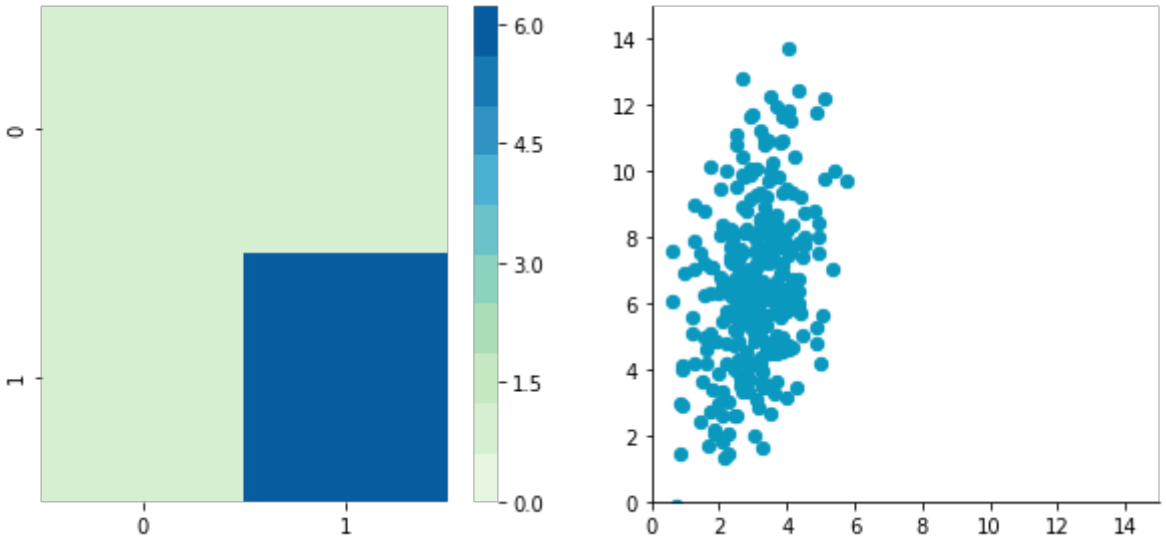
where $\boldsymbol{X}'$ is the standardized dataset, $\boldsymbol{X}$ the original dataset, x the mean of $\boldsymbol{X}$ and $\sigma_{\boldsymbol{X}}$ the standard deviation of $\boldsymbol{X}$.

```
def standardize(X):
    newX = center(X)/np.std(X, axis = 0)
    return newX
```

Let's create another dataset with a different scale to check that it is working.

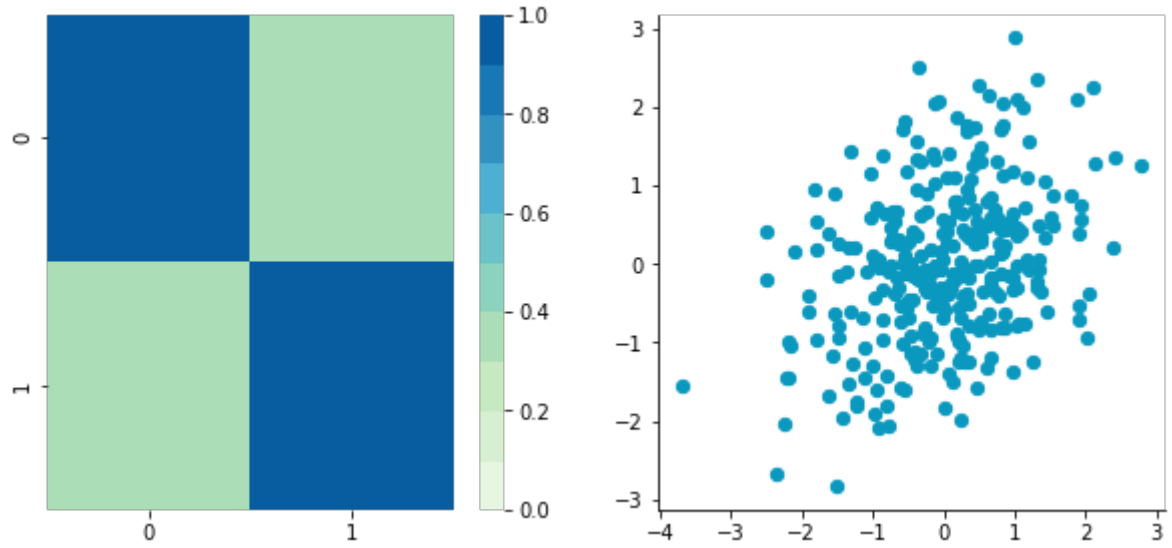
```
np.random.seed(1234)
c1 =  np.random.normal(3, 1, 300)
c2 = c1 + np.random.normal(7, 5, 300)/2.
C = np.array([c1, c2]).T

plotDataAndCov(C)
plt.xlim(0, 15)
plt.ylim(0, 15)
plt.show()
plt.close()
```

```
Covariance matrix:
[[ 0.95171641  0.83976242]
 [ 0.83976242  6.22529922]]
```



We can see that the scales of x and y are different. Note also that the correlation seems smaller because of the scale differences. Now let's standardise it:

```
CStandardized = standardize(C)

plotDataAndCov(CStandardized)
plt.show()
plt.close()
```



Looks good! You can see that the scales are the same and that the dataset is zero-centered according to both axes. Now, have a look at the covariance matrix: you can see that the variance of each coordinate (the top-left cell and the bottom-right cell) is equal to 1. By the way, this new covariance matrix is actually the correlation matrix!💥 The Pearson correlation coefficient between the two variables ($\boldsymbol{c}_1$ and $\boldsymbol{c}_2$) is 0.54220151.

C. Whitening

Whitening or sphering data means that we want to transform it in a way to have a covariance matrix that is the identity matrix (1 in the diagonal and 0 for the other cells; more details on the identity matrix). It is called whitening in reference to white noise.

We now have all the tools that we need to do it. It involves the following steps:

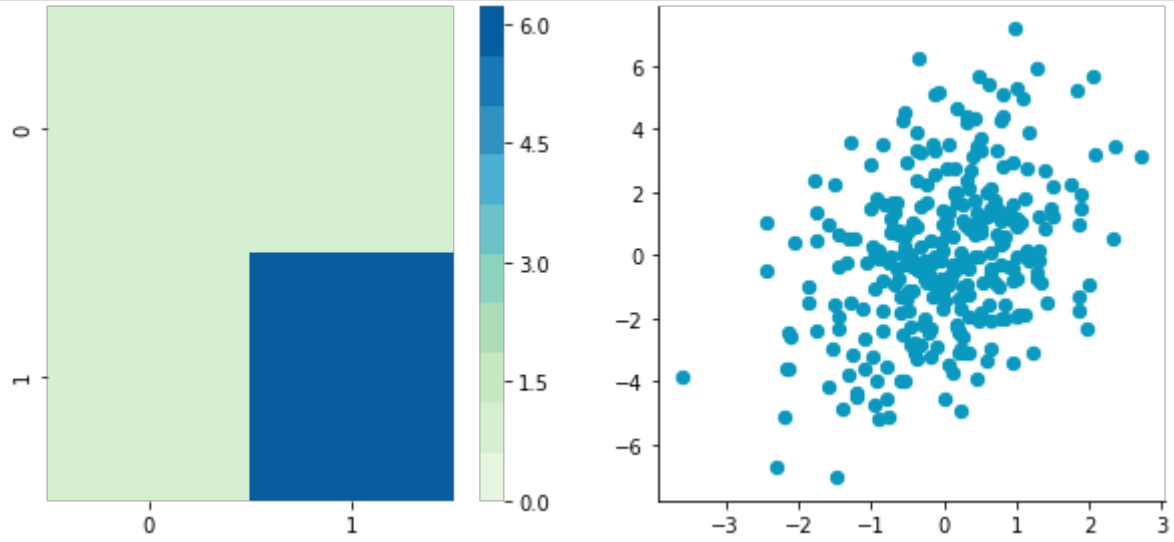
```
1- Zero-center the data
2- Decorrelate the data
3- Rescale the data
```

Let's take again $\boldsymbol{C}$ and try to do these steps.

1. Zero-centering

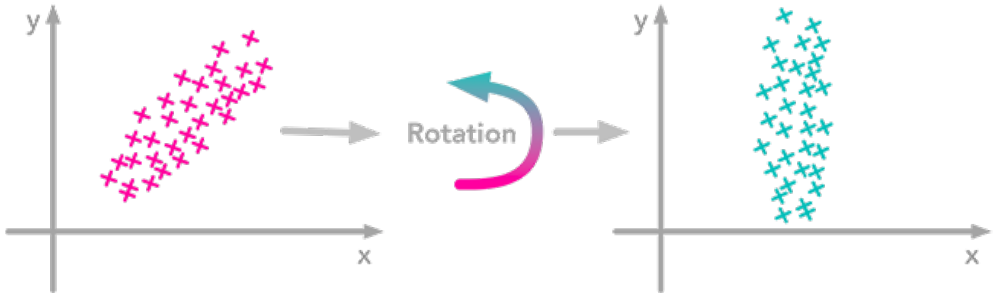
```
CCentered = center(C)

plotDataAndCov(CCentered)
plt.show()
plt.close()
```

```
Covariance matrix:
[[ 0.95171641  0.83976242]
```



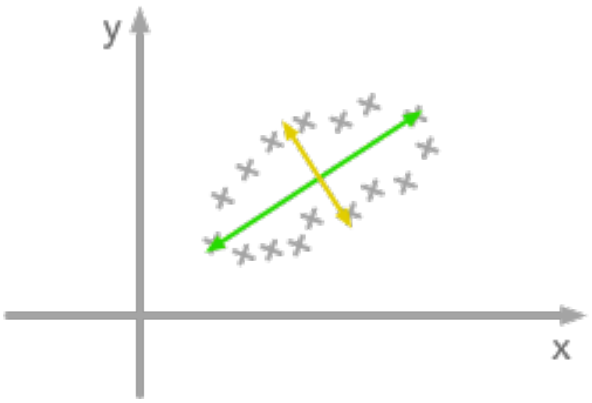
2. Decorrelate

At this point, we need to decorrelate our data. Intuitively, it means that we want to rotate the data until there is no correlation anymore. Look at the following cartoon to see what I mean:



The left plot shows correlated data. For instance, if you take a data point with a big x value, chances are that y will also be quite big. Now take all data points and do a rotation (maybe around 45 degrees counterclockwise): the new data (plotted on the right) is not correlated anymore.

The question is: how could we find the right rotation in order to get the uncorrelated data? Actually, it is exactly what the **eigenvectors** of the covariance matrix do: they indicate the direction where the spread of the data is at its maximum:



The eigenvectors of the covariance matrix give you the direction that maximizes the variance. The direction of the green line is where the variance is maximum. Just look at the smallest and largest

☰

For more details about the eigendecomposition, see this post.

So we can decorrelate the data by projecting it on the eigenvectors basis. This will have the effect to apply the rotation needed and remove correlations between the dimensions. Here are the steps:

```
1- Calculate the covariance matrix
2- Calculate the eigenvectors of the covariance matrix
3- Apply the matrix of eigenvectors to the data (this will apply the rotation)
```

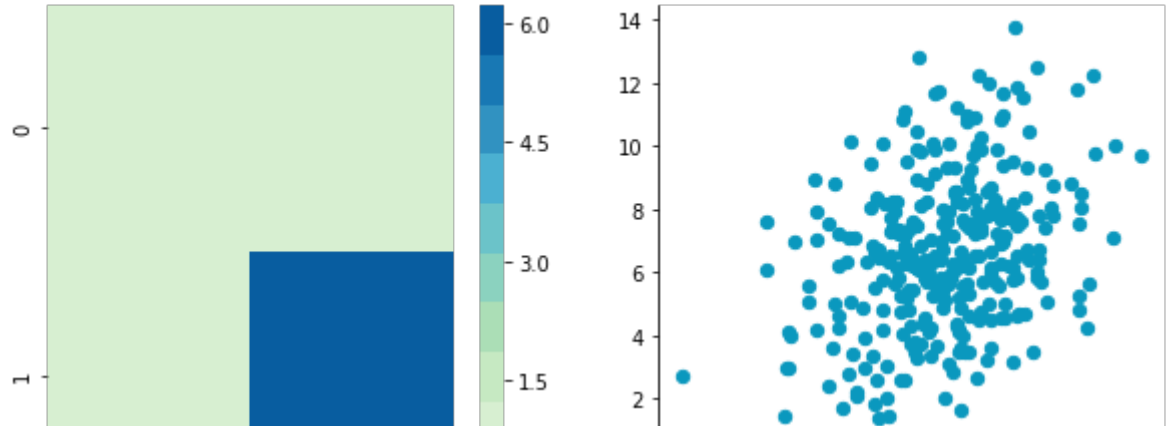
Let's pack that into a function:

```
def decorrelate(X):
    XCentered = center(X)
    cov = XCentered.T.dot(XCentered)/float(XCentered.shape[0])
    # Calculate the eigenvalues and eigenvectors of the covariance matrix
    eigVals, eigVecs = np.linalg.eig(cov)
    # Apply the eigenvectors to X
    decorrelated = X.dot(eigVecs)
    return decorrelated
```

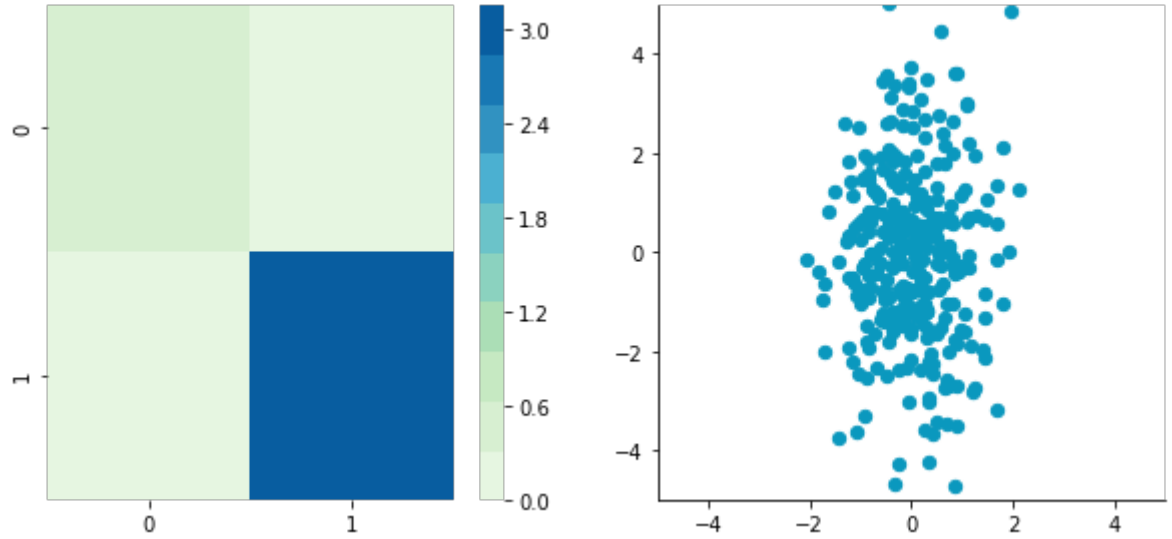
Let's try to decorrelate our zero-centered matrix $\boldsymbol{C}$ to see it in action:

```
plotDataAndCov(C)
plt.show()
plt.close()

CDecorrelated = decorrelate(C)
plotDataAndCov(CDecorrelated)
plt.xlim(-5,5)
plt.ylim(-5,5)
plt.show()
plt.close()
```

```
Covariance matrix:
[[ 0.95171641  0.83976242]
 [ 0.83976242  6.22529922]]
```



```
Covariance matrix:
[[  5.96126981e-01  -1.48029737e-16]
 [ -1.48029737e-16   3.15205774e+00]]
```



Nice! This is working 🎄

We can see that the correlation is not here anymore and that the covariance matrix (now a diagonal matrix) confirms that the covariance between the two dimensions is equal to 0.

3. Rescale the data

The next step is to scale the uncorrelated matrix in order to obtain a covariance matrix corresponding to the identity matrix (ones on the diagonal and zeros on the other cells). To do that we scale our decorrelated data by dividing each dimension by the square-root of its corresponding eigenvalue.

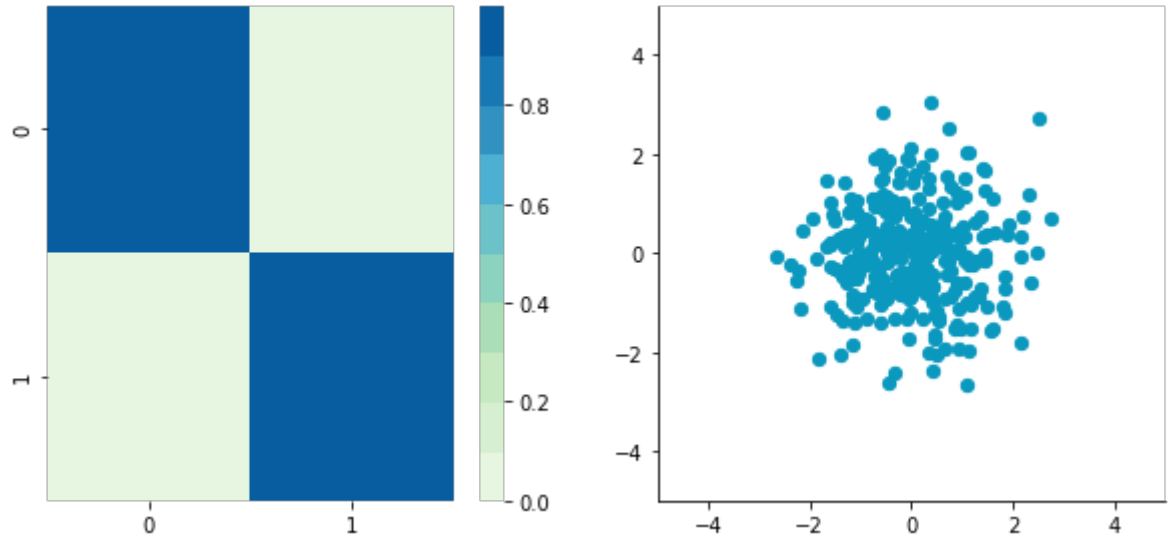
```
def whiten(X):
    XCentered = center(X)
    cov = XCentered.T.dot(XCentered)/float(XCentered.shape[0])
    # Calculate the eigenvalues and eigenvectors of the covariance matrix
    eigVals, eigVecs = np.linalg.eig(cov)
    # Apply the eigenvectors to X
    decorrelated = X.dot(eigVecs)
    # Rescale the decorrelated data
    whitened = decorrelated / np.sqrt(eigVals + 1e-5)
    return whitened
```

Note: we add a small value (here 10^{-5}) to avoid the division by 0.

```
CWhitened = whiten(C)

plotDataAndCov(CWhitened)
```

```
plt.close()
```

```
Covariance matrix:
[[  9.99983225e-01  -1.06581410e-16]
 [ -1.06581410e-16   9.99996827e-01]]
```



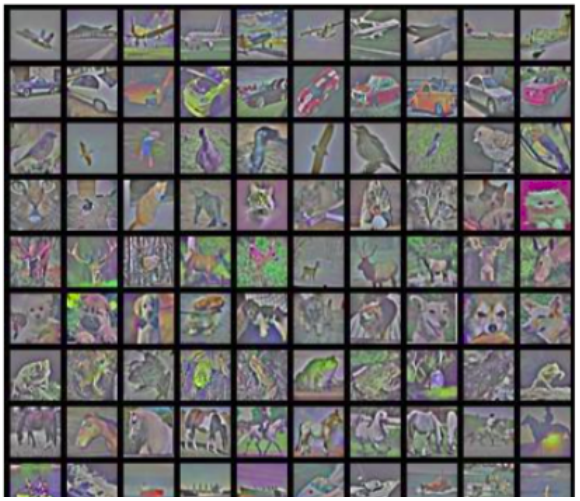
Hooray! We can see that with the covariance matrix that this is all good. We have something that really looks to the identity matrix (1 on the diagonal and 0 elsewhere). 🌵

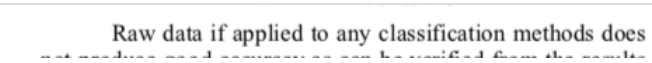
3. Image whitening

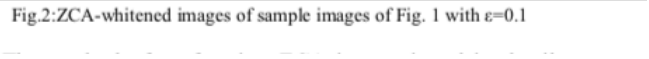
We will see how whitening can be applied to preprocess image dataset. To do so we will use the paper of Pal & Sudeep (2016) where they give some details about the process. This preprocessing technique is called Zero component analysis (ZCA).

Check out the paper, but here is the kind of result they got:

training data by Krizhevsky [13] and later by Coates et al. [15]. The ZCA transformation makes the edges of the objects more prominent as can be seen from fig. 2. The convolutional layers detect various features through the feature maps based on these edges.







Whitening images from the CIFAR10 dataset. Results from the paper of Pal & Sudeep (2016). The original images (left) and the images after the ZCA (right) are shown.

First thing first: we will load images from the CIFAR dataset. This dataset is available from Keras but you can also download it here.

```
from keras.datasets import cifar10

(X_train, y_train), (X_test, y_test) = cifar10.load_data()

X_train.shape
```

```
(50000, 32, 32, 3)
```

The training set of the CIFAR10 dataset contains 50000 images. The shape of `X_train` is (50000, 32, 32, 3). Each image is 32px by 32px and each pixel contains 3 dimensions (R, G, B). Each value is the brightness of the corresponding color between 0 and 255.

We will start by selecting only a subset of the images, let's say 1000:

```
X = X_train[:1000]
print X.shape
```

```
(1000, 32, 32, 3)
```

That's better! Now we will reshape the array to have flat image data with one image per row. Each image will be (1, 3072) because $32 \times 32 \times 3 = 3072$. Thus, the array containing all images will be (1000, 3072):

```
X = X.reshape(X.shape[0], X.shape[1]*X.shape[2]*X.shape[3])
print X.shape
```

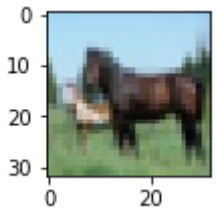
```
(1000, 3072)
```

The next step is to be able to see the images. The function `imshow()` from Matplotlib (doc) can be used to show images. It needs images with the shape ($M \times N \times 3$) so let's create a function to reshape the images and be able to visualize them from the shape (1, 3072).

```
def plotImage(X):
    plt.figure(figsize=(1.5, 1.5))
    plt.imshow(X.reshape(32,32,3))
    plt.show()
```

For instance, let's plot one of the images we have loaded:

```
plotImage(X[12, :])
```



Cute! 🌵

We can now implement the whitening of the images. Pal & Sudeep (2016) describe the process:

1. The first step is to rescale the images to obtain the range [0, 1] by dividing by 255 (the maximum value of the pixels).

Remind that the formula to obtain the range [0, 1] is:

$$\frac{data - min(data)}{max(data) - min(data)}$$

but here, the minimum value is 0, so this leads to:

$$\frac{data}{max(data)} = \frac{data}{255}$$

```
X_norm = X / 255.
print 'X.min()', X_norm.min()
print 'X.max()', X_norm.max()
```

```
X.min() 0.0
X.max() 1.0
```

Mean subtraction: per-pixel or per-image?

Ok cool, the range of our pixel values is between 0 and 1 now. The next step is:

1. Subtract the mean from all image.

Be careful here 🌪️📈:

One way to do it is to take each image and remove the mean of this image from every pixel (Jarrett et al., 2009). The intuition behind this process is that it centers the pixels of each image around 0.

Another way to do it is to take each of the 3072 pixels that we have (32 by 32 pixels for R, G and

☰

you will feed your network with the images, each pixel is considered as a different feature. With the per-pixel mean subtraction, we have centered each feature (pixel) around 0. This technique is commonly used (e.g Wan et al., 2013).

We will now do the per-pixel mean subtraction from our 1000 images. Our data are organized with these dimensions (images, pixels). It was (1000, 3072) because there are 1000 images with $32 \times 32 \times 3 = 3072$ pixels. The mean per-pixel can thus be obtained from the first axis:

```
X_norm.mean(axis=0).shape
```

```
(3072,)
```

This gives us 3072 values which is the number of means: one per pixel. Let's see the kind of values we have:

```
X_norm.mean(axis=0)
```

```
array([ 0.5234    ,  0.54323137,  0.5274    , ...,  0.50369804,
        0.50011765,  0.45227451])
```

This is near 0.5 because we already have normalized to the range [0, 1]. However, we still need to remove the mean from each pixel:

```
X_norm = X_norm - X_norm.mean(axis=0)
```

Just to convince ourselves that it worked, we will compute the mean of the first pixel. Let's hope that it is 0.

```
X_norm.mean(axis=0)
```

```
array([ -5.30575583e-16,  -5.98021632e-16,  -4.23439062e-16, ...,
        -1.81965554e-16,  -2.49800181e-16,   3.98570066e-17])
```

This is not exactly 0 but it is small enough that we can consider that it worked! 🌵

Now we want to calculate the covariance matrix of the zero-centered data. Like we have seen above, we can calculate it with the `np.cov()` function from Numpy.

There are two possible correlation matrices that we can calculate from the matrix $\boldsymbol{X}$: either the correlation between rows or between columns. In our case, each row of the matrix $\boldsymbol{X}$ is an image, so the rows of the matrix correspond to the observations and the columns of the matrix corresponds to the features (the images pixels). We want to calculate the correlation between the

To do so, we will tell this to Numpy with the parameter `rowvar=False` (see the doc): it will use the columns as variables (or features) and the rows as observations.

```
cov = np.cov(X_norm, rowvar=False)
```

The covariance matrix should have a shape of 3072 by 3072 to represent the correlation between each pair of pixels (and there are 3072 pixels):

```
cov.shape
```

```
(3072, 3072)
```

Now the magic part: we will calculate the singular values and vectors of the covariance matrix and use them to rotate our dataset. Have a look at my post on the singular value decomposition if you need more details!

```
U,S,V = np.linalg.svd(cov)
```

In the paper, they used the following equation:

$$\boldsymbol{X}_{ZCA} = \boldsymbol{U}.diag(\frac{1}{\sqrt{diag(\boldsymbol{S})+\epsilon}}).\boldsymbol{U}^{\mathrm{T}}.\boldsymbol{X}$$

with $\boldsymbol{U}$ the left singular vectors, and $\boldsymbol{S}$ the singular values of the covariance of the initial normalized dataset of images and $\boldsymbol{X}$ the normalized dataset. ϵ (*epsilon*) is an hyper-parameter called the whitening coefficient. $diag(a)$ corresponds to a matrix with the vector a as a diagonal and 0 in all other cells.

We will try to implement this equation. Let's start by checking the dimensions of the SVD:

```
print U.shape, S.shape
```

```
(3072, 3072) (3072,)
```

$\boldsymbol{S}$ is a vector containing 3072 elements (the singular values). $diag(\boldsymbol{S})$ will thus be of shape (3072, 3072) with $\boldsymbol{S}$ as the diagonal:

```
print np.diag(S)
print '\nshape:', np.diag(S).shape
```

```
[  0.00000000e+00   4.08234845e+00   0.00000000e+00 ...,   0.00000000e+00
   0.00000000e+00   0.00000000e+00]
[  0.00000000e+00   0.00000000e+00   2.41075267e+00 ...,   0.00000000e+00
   0.00000000e+00   0.00000000e+00]
...,
[  0.00000000e+00   0.00000000e+00   0.00000000e+00 ...,   3.92727365e-05
   0.00000000e+00   0.00000000e+00]
[  0.00000000e+00   0.00000000e+00   0.00000000e+00 ...,   0.00000000e+00
   3.52614473e-05   0.00000000e+00]
[  0.00000000e+00   0.00000000e+00   0.00000000e+00 ...,   0.00000000e+00
   0.00000000e+00   1.35907202e-15]]

shape: (3072, 3072)
```

$diag(\frac{1}{\sqrt{diag(\boldsymbol{S})+\epsilon}})$ is also of shape (3072, 3072) as well as $\boldsymbol{U}$ and $\boldsymbol{U}^{\mathrm{T}}$. We have seen also that $\boldsymbol{X}$ has the shape (1000, 3072) and we need to transpose it to have (3072, 1000). The shape of $\boldsymbol{X}_{ZCA}$ is thus:

$$(3072, 3072).(3072, 3072).(1000, 3072)^{\mathrm{T}} = (3072, 3072).(3072, 1000) = (3072, 1000)$$

which corresponds to the shape of the initial dataset after transposition. Nice!

We have:

```
epsilon = 0.1
X_ZCA = U.dot(np.diag(1.0/np.sqrt(S + epsilon))).dot(U.T).dot(X_norm.T).T
```

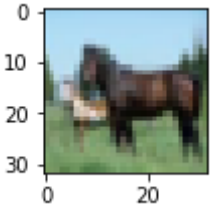
Let's rescale the images:

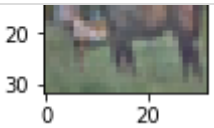
```
X_ZCA_rescaled = (X_ZCA - X_ZCA.min()) / (X_ZCA.max() - X_ZCA.min())
print 'min:', X_ZCA_rescaled.min()
print 'max:', X_ZCA_rescaled.max()
```

```
min: 0.0
max: 1.0
```

Finally, we can look at the effect of whitening by comparing an image before and after whitening:

```
plotImage(X[12, :])
plotImage(X_ZCA_rescaled[12, :])
```





Hooray! That's great!⚡It looks like the images from the paper Pal & Sudeep (2016). They used `epsilon = 0.1`. You can try other values to see the effect on the image.

That's all! 🌴

I hope that you found something interesting in this article!

You can fork the Jupyter notebook on Github here!

References

K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?," in 2009 IEEE 12th International Conference on Computer Vision, 2009, pp. 2146–2153.

A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Master's thesis, University of Tront, 2009.

Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient BackProp," in Neural Networks: Tricks of the Trade, Springer, Berlin, Heidelberg, 2012, pp. 9–48.

K. K. Pal and K. S. Sudeep, "Preprocessing for image classification by convolutional neural networks," in 2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT), 2016, pp. 1778–1781.

L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of Neural Networks using DropConnect," in International Conference on Machine Learning, 2013, pp. 1058–1066.

And also these great resources and QA:

Wikipedia - Whitening transformation

CS231 - Convolutional Neural Networks for Visual Recognition

Dustin Stansbury - The Clever Machine

Some details about the covariance matrix

SO - Image whitening in Python

Mean normalization per image or from the entire dataset

Mean subtraction - all images or per image?

☰

Kaggle kernel on ZCA

How ZCA is implemented in Keras

✠ Previous

Deep Learning Book Series · 2.12 Example Principal Components Analysis

✠ Next

Deep Learning Book Series 3.1 to 3.3 Probability Mass and Density Functions