In [2]:
```
import numpy as np
import pandas as pd
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, Conv2DTranspose
from keras.layers import BatchNormalization, Reshape, LeakyReLU, Dropout
from keras.optimizers import Adam
from sklearn.decomposition import PCA
from time import time
import matplotlib.pyplot as plt
%matplotlib inline
```

In [3]:
```
X_train = pd.read_csv("./data/ClustREFGenes-master/Data/Core_genome/Data_Core_Genome_Ecoli_log2.c
                      index_col=0)
print("Dimensionalidade dos dados: ", X_train.shape)
X_train.head()
```

```
Dimensionalidade dos dados:  (4051, 9)
```

Out[3]:

	BB9	BB10	BB17	BB19	BB20	BB21	BB11	BB12	BB18
Genes									
accD	6.875411	7.047582	7.431765	7.105877	6.516094	6.676126	6.304694	6.168221	6.245553
aceF	7.732412	7.674997	8.397717	7.455056	7.277269	6.525536	7.455730	6.403830	7.597941
ackA	7.231720	7.260976	8.033280	6.921924	6.920829	6.556644	6.358150	5.888768	6.359310
agaV	6.048825	6.250033	5.120269	5.559767	5.915593	6.279490	6.441998	6.553099	6.105364
alaS	7.811728	7.853890	8.622037	7.636451	7.641365	7.125920	7.164957	6.555678	7.098590

PCA:

- para ver a distribuição dos dados, estes serão reduzidos à só dois dimemsões com PCA.

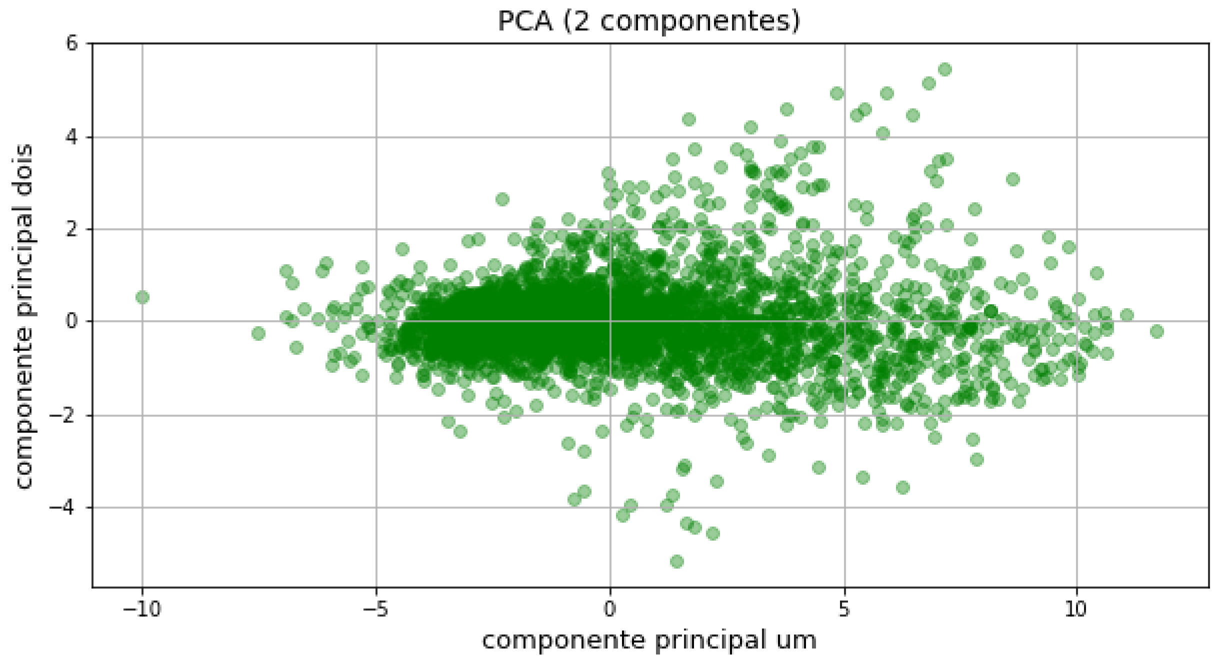
In [4]:
```
pca = PCA(n_components=2)
pca.fit(X_train)
```

Out[4]:
```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

In [5]:
```
X_pca = pca.transform(X_train)
print("Dimensionalidade: ", X_pca.shape)
```

```
Dimensionalidade:  (4051, 2)
```

In [6]:
```
plt.figure(figsize=(10,5))
plt.title("PCA (2 componentes)", fontsize=14)
plt.xlabel("componente principal um", fontsize=13)
plt.ylabel("componente principal dois", fontsize=13)
plt.grid()
plt.scatter(X_pca[:,0], X_pca[:,1], color="green", alpha=.4);
```



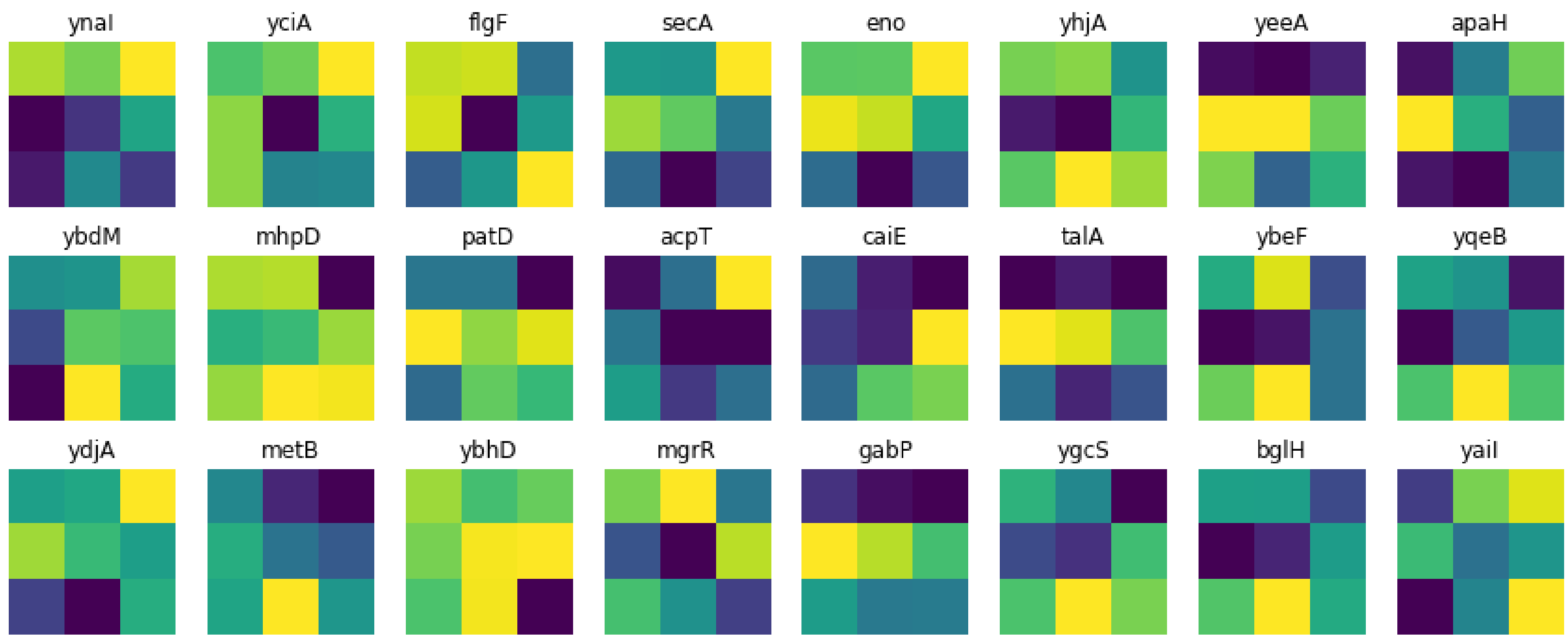
- vou trocar a dimensionalidade de cada gen, por uma dimesionalidade de 3x3, para assim ver o gen como uma imagem.

In [7]:
```
X_img = np.array(X_train).reshape((X_train.shape[0],3,3,1))
X_img = (X_img - np.mean(X_img))/np.mean(X_img)
X_lab = X_train.index
```

In [9]:

```
idx = np.random.randint(low=0, high=X_train.shape[0], size=24)
imgs =  X_img[idx]
titles = X_lab[idx]
fig = plt.figure(figsize=(15,6))
p=0
#plt.title("Genomas representados na forma de uma imagem", fontsize=12)
plt.axis("off");
print("----------------------------- Genomas representados na forma de uma matriz --------------
for i in imgs:
    ax=fig.add_subplot(3,8,p+1)
    plt.title(titles[p])
    plt.imshow(i.reshape(3,3))
    plt.axis("off");
    p += 1
#plt.colorbar(ax=ax);
```

```
----------------------------- Genomas representados na forma de uma matriz -------------------------
----
```



In [ ]:

In [10]:

```
class GANs():
    #inialização dos parâmetros
    def __init__(self, width, height, channels, noise_input):
        self.width = width
        self.height = height
        self.channels = channels
        self.dim = (self.width, self.height, self.channels)
        self.noise_input = noise_input
        self.g_loss = []
        self.d_loss = []
        self.g_lpe = []
        self.d_lpe = []
        #self.optimizer = Adam(lr=0.0001, beta_1=0.5)
        self.optimizerD = Adam(lr=0.0001, beta_1=0.5)
        self.optimizerG = Adam(lr=0.0004, beta_1=0.5)
        self.G = self.noise_generator()
        print("Compilando o gerador...")
        self.G.compile(loss='binary_crossentropy', optimizer=self.optimizerG)
        self.D = self.discriminator()
        print("Compilando o discriminador...")
        self.D.compile(loss='binary_crossentropy', optimizer=self.optimizerD, metrics=['accuracy
        self.stacked_generator_discriminator = Sequential()
        self.stacked_generator_discriminator.add(self.G)
        self.stacked_generator_discriminator.add(self.D)
        self.D.trainable = False
        self.stacked_generator_discriminator.compile(loss='binary_crossentropy', optimizer=self.

    #criação do gerador de imagens fake
    def noise_generator(self):
        model = Sequential()
        model.add(Dense(16, input_shape=(self.noise_input,)))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dense(32))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dense(64))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dense(self.width*self.height*self.channels, activation="tanh"))
        model.add(Reshape((self.width, self.height, self.channels)))
        return model

    #criação do discriminador
    def discriminator(self):
```

```
        model = Sequential()
        model.add(Dense(64, input_shape=self.dim))
        model.add(LeakyReLU(alpha=0.2)) #función rectificadora
        model.add(Dense(32))
        model.add(LeakyReLU(alpha=0.2)) #función rectificadora
        model.add(Dense(16))
        model.add(Flatten())
        model.add(Dense(1, activation='sigmoid'))

        return model

    #Para obter o sumary do gerador
    def summary_gerador(self):
        return self.G.summary()

    #Para obter o sumary do gerador
    def summary_discriminador(self):
        return self.D.summary()

    #pra obter os batches pra o treino
    def get_batches(self, X_train, batch_size):
        """
        X_train: dataset para o treino
        epochs: quantidade de epocas para o treino do gradiente
        batch: tamanho to batch pra o treino de cada epochs
        """
        batches = []
        num_bat = int(np.ceil(X_train.shape[0]/batch_size))
        lim_i = 0
        lim_s = batch_size
        for i in range(num_bat):
            if lim_s > X_train.shape[0]:
                lim_s = X_train.shape[0]
            batches.append(X_train[lim_i:lim_s])
            lim_i += batch_size
            lim_s += batch_size

        return batches

    #devolve o loss do gerador e do discriminador
    def get_loss(self):
        return [self.g_loss, self.d_loss]

```

```
    #treinamento da GAN
    def train(self, X_train, epochs, batch_size):
        self.d_loss = []
        self.g_loss = []
        for cnt in range(epochs):
            batches = self.get_batches(X_train, batch_size)
            count_b = 0
            t_i = time()
            for batch in batches:
                gen_noise = np.random.normal(0, 1, (np.int64(batch.shape[0]), self.noise_input))
                #gerando as imagens fake
                syntetic_images = self.G.predict(gen_noise)
                #criação do array de treinamento
                x_combined_batch = np.concatenate((batch, syntetic_images))
                y_combined_batch = np.concatenate((np.ones((batch.shape[0], 1)),
                                                   np.zeros((batch.shape[0], 1))))
                #treino do discriminador
                d_l = self.D.train_on_batch(x_combined_batch, y_combined_batch)
                self.d_loss.append(d_l[0])
                # train generator
                noise = np.random.normal(0, 1, (batch.shape[0], self.noise_input))
                y_mislabled = np.ones((batch.shape[0], 1))

                g_l = self.stacked_generator_discriminator.train_on_batch(noise, y_mislabled)
                self.g_loss.append(g_l)
                count_b += 1
                if (count_b%20)==0:
                    t_f = time()
                    t = t_f - t_i
                    t_i = time()
                    print ('epoch:[%d/%d] batch:[%d/%d], [Discriminator::d_loss: %f], [Generator
                           % (cnt+1,epochs,count_b,len(batches),d_l[0],g_l,t))
            self.g_lpe.append(g_l)
            self.d_lpe.append(d_l[0])
```

In [11]:

```
gan = GANs(width=3, height=3, channels=1, noise_input=100)
```

```
Compilando o gerador...
Compilando o discriminador...
```

In [12]:
```
img_p = gan.G.predict(np.random.normal(0,1,(1,100)))
plt.imshow(img_p.reshape(3,3));
plt.axis("off");
```

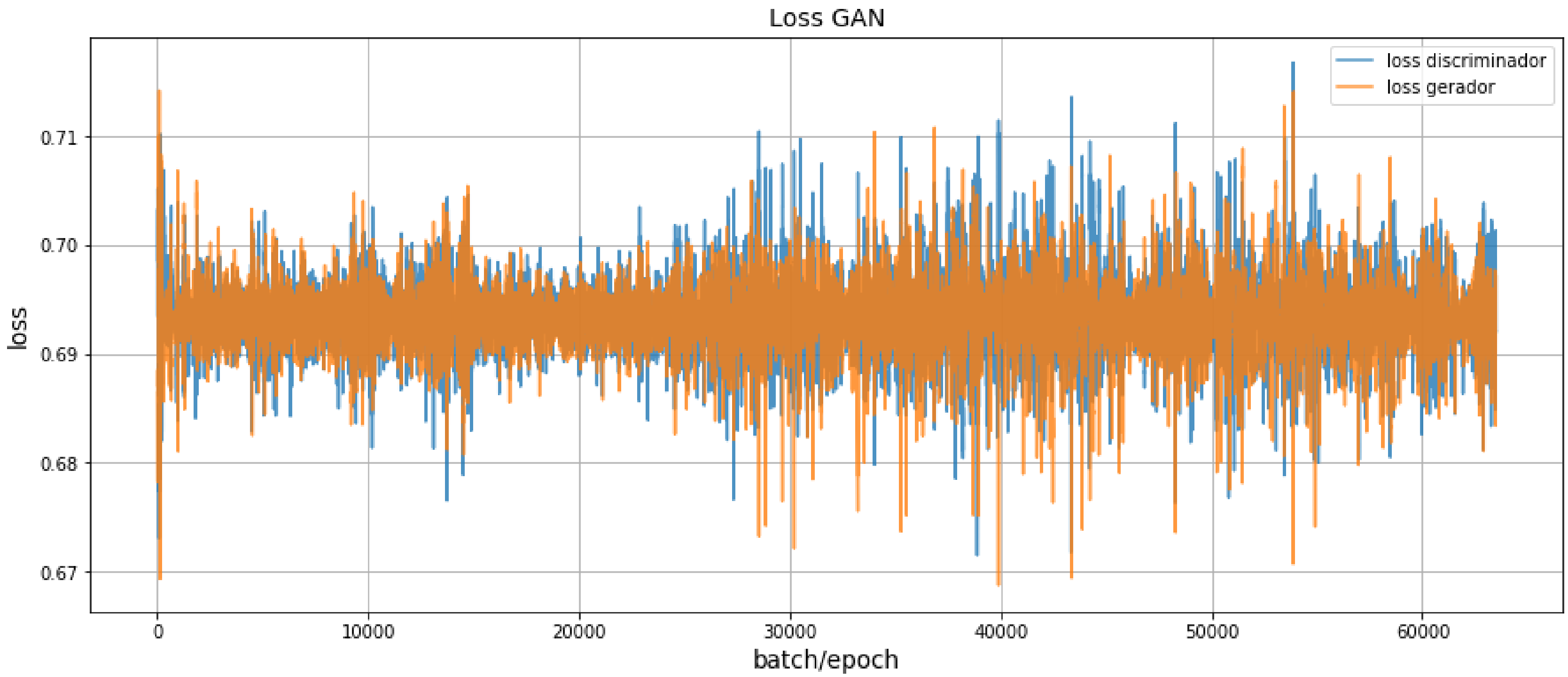
In [ ]:
```
t_i = time()
gan.train(X_img, epochs=500,batch_size=32)
t_f = time()
```

In [14]:
```
print("tempo de execução: ", (t_f-t_i)/60, "[min]")
```

```
tempo de execução:  3.494025520483653 [min]
```

In [15]:
```
g_loss, d_loss = gan.get_loss()

plt.figure(figsize=(15,6))
#plt.plot(range(len(g_loss)), g_loss)
plt.title("Loss GAN", fontsize=14)
plt.ylabel("loss", fontsize=13.5)
plt.xlabel("batch/epoch", fontsize=13.5)
plt.plot(range(np.array(d_loss).shape[0]), np.array(d_loss), label="loss discriminador", alpha=.8
plt.plot(range(np.array(g_loss).shape[0]), np.array(g_loss), label="loss gerador", alpha=.8)
#plt.plot(range(np.array(g_loss).shape[0]), 0.5*np.ones(np.array(g_loss).shape[0]),
#         color="black", label="objetivo", linestyle='--')
plt.grid()
#plt.yticks([0, 0.5,1,1.5,2,2.5,3,3.5,4,4.5,5])
plt.legend();
```



In [16]:

```
plt.figure(figsize=(15,6))
plt.title("Loss GAN", fontsize=14)
plt.ylabel("loss", fontsize=13.5)
plt.xlabel("epochs", fontsize=13.5)
plt.plot(range(len(gan.d_lpe)), np.array(gan.d_lpe), label="loss discriminador")
plt.plot(range(len(gan.g_lpe)), np.array(gan.g_lpe), label="loss gerador")
#plt.plot(range(np.array(g_loss).shape[0]), 0.5*np.ones(np.array(g_loss).shape[0]),
#         color="black", label="objetivo", linestyle='--')
plt.grid()
plt.legend();
```

In [ ]:

```
 
```

In [17]:

```
num_imgs = 24 #número de imágenes a mostrar aleatoriamente
img_pre = gan.G.predict(np.random.normal(0,1,(num_imgs,100)))
fig = plt.figure(figsize=(15,6))
for i in range(num_imgs):
    ax=fig.add_subplot(3,8,i+1)
    img = img_pre[i]
    plt.imshow(img.reshape((3,3)))
    ax.axis("off")
plt.show()
```

In [22]:

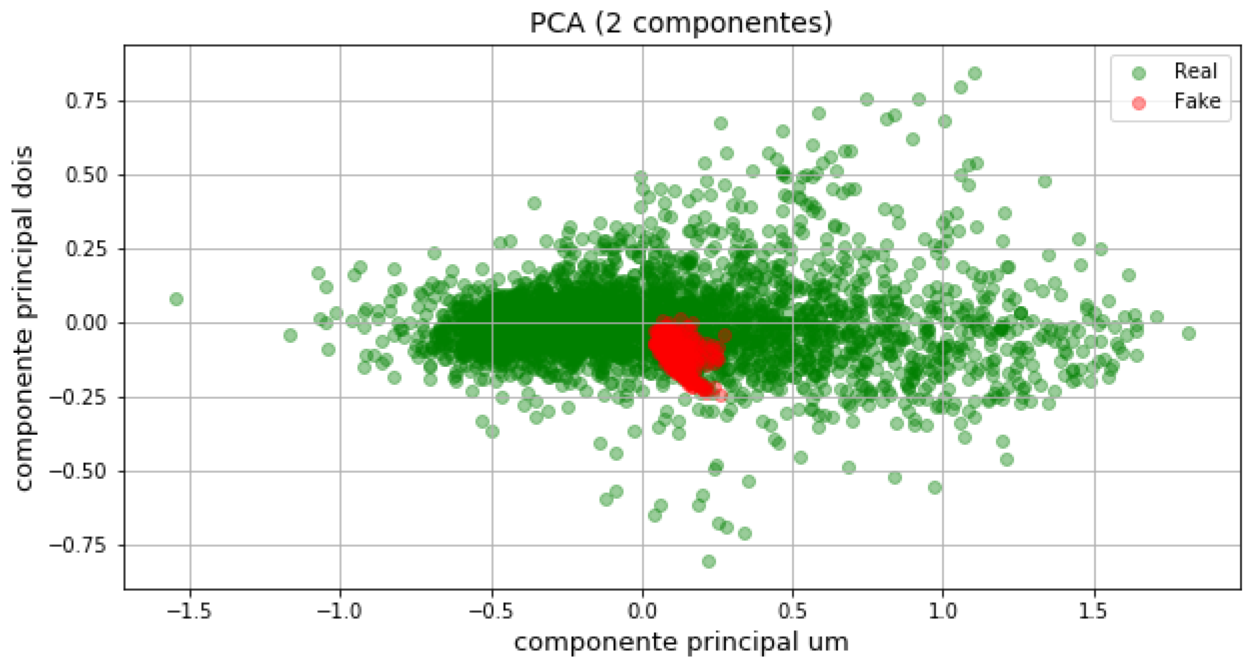
```
num_imgs = 500
fakes = gan.G.predict(np.random.normal(0,1,(num_imgs,100)))
print("imagens fake: ", fakes.shape)
fakes = fakes.reshape(num_imgs,9)
print("re-dimesionalidade: ", fakes.shape)
pca2 = PCA(n_components=2)
pca2.fit(X_img.reshape(X_img.shape[0],9))

X_real = pca2.transform(X_img.reshape(X_img.shape[0],9))
X_fake = pca2.transform(fakes)
```

```
imagens fake:  (500, 3, 3, 1)
re-dimesionalidade:  (500, 9)
```

In [23]:

```
plt.figure(figsize=(10,5))
plt.title("PCA (2 componentes)", fontsize=14)
plt.xlabel("componente principal um", fontsize=13)
plt.ylabel("componente principal dois", fontsize=13)
plt.grid()
plt.scatter(X_real[:,0], X_real[:,1], color="green", alpha=.4, label="Real")
plt.scatter(X_fake[:,0], X_fake[:,1], color="red", alpha=.4, label="Fake")
plt.legend();
```



In [ ]:

```

```