In [11]:
```
import numpy as np
import pandas as pd
import seaborn as sns
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, SGD, RMSprop
from sklearn.decomposition import PCA
from sklearn.utils import shuffle
from PIL import Image
from time import time
import matplotlib.pyplot as plt
%matplotlib inline
```

In [12]:
```
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[12]:

Genes	BB9	BB10	BB17	BB19	BB20	BB21	BB11	BB12	BB18
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 dimensões com PCA.

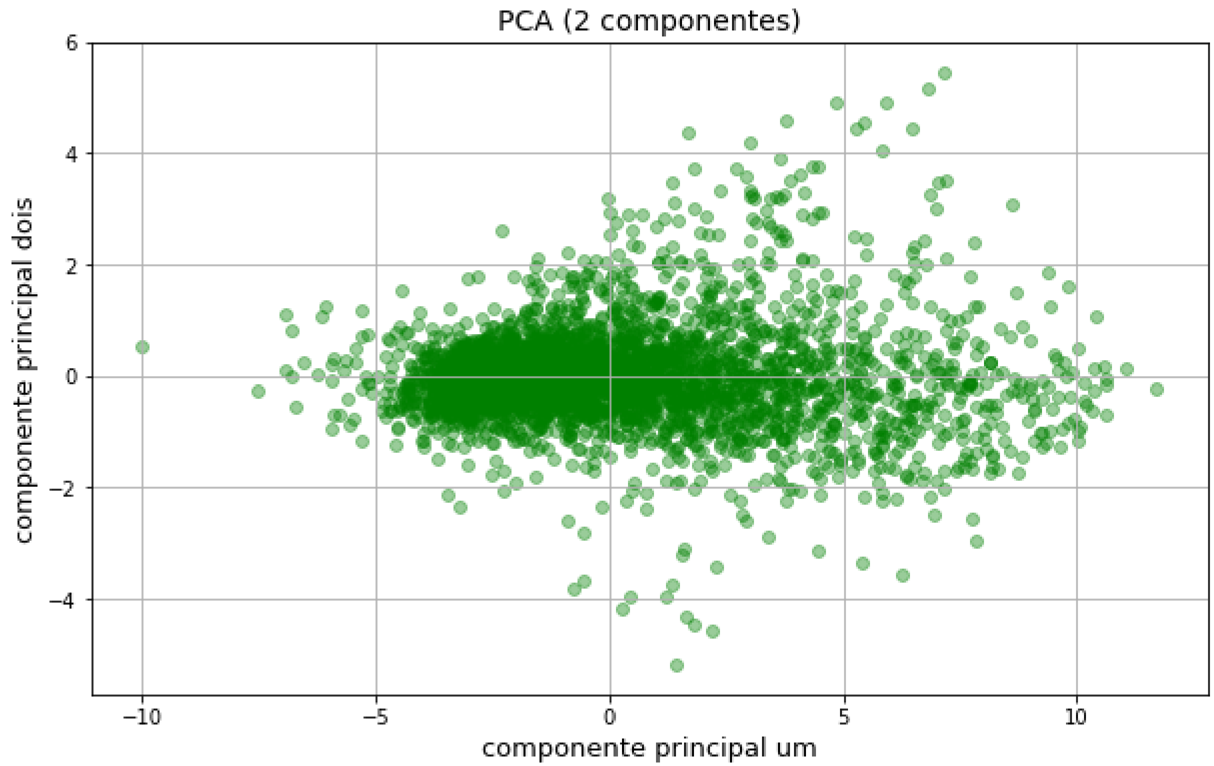
In [13]:
```
pca = PCA(n_components=2)
pca.fit(X_train)
```

Out[13]:
```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

In [14]:
```
X_pca = pca.transform(X_train)
print("Dimensionalidade: ", X_pca.shape)
```

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

In [15]:
```
plt.figure(figsize=(10,6))
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);
```



Normalização dos dados

- Para poder fazer uso das GAN's, a gente tem que normalizar os dados, para eso é usada a seguinte normalização:

$$X = \frac{x_i - mean(X)}{mean(X)}$$

In [16]:
```
X_train = (X_train - np.mean(X_train))/np.mean(X_train)
```

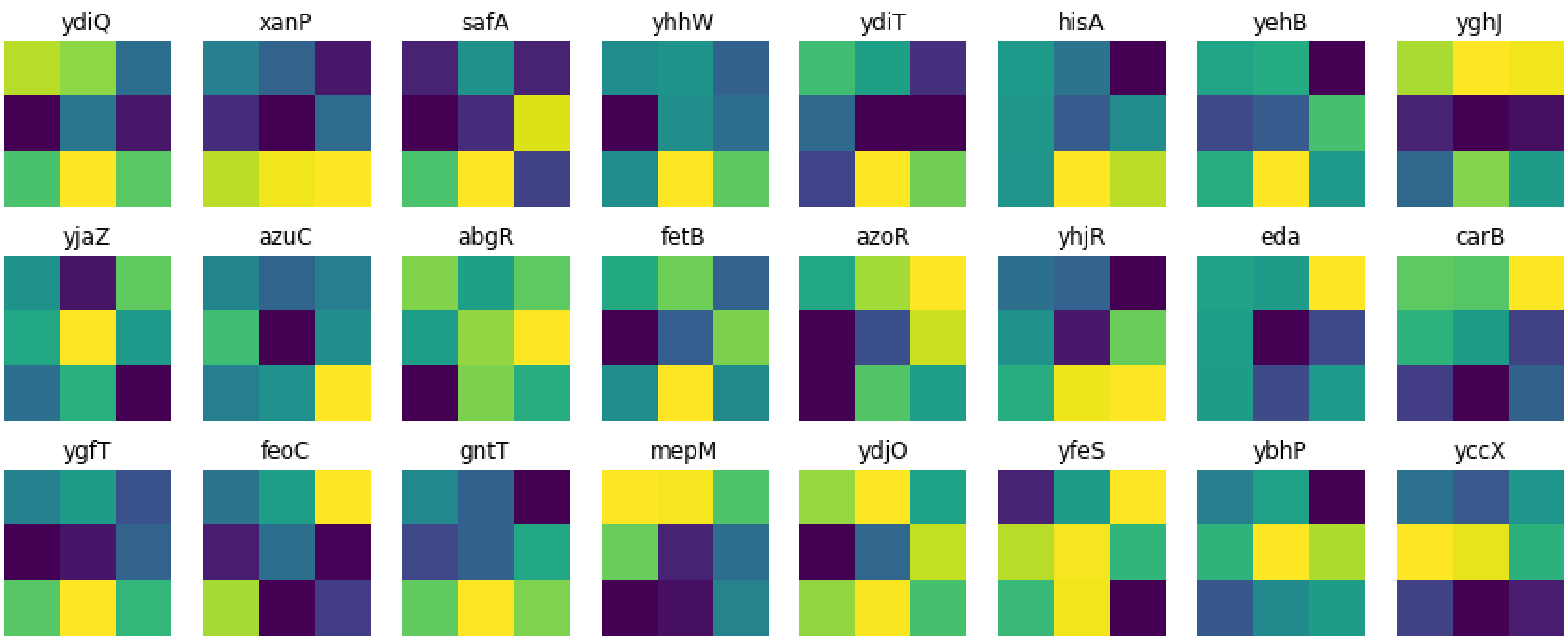
- vou trocar a dimensionalidade de cada gen, por uma dimesionalidade de 3x3, para assim ver o gen como uma matrix.

In [17]:
```
X_img = np.array(X_train).reshape((X_train.shape[0],3,3,1))
X_lab = X_train.index
```

In [18]:

```
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
```

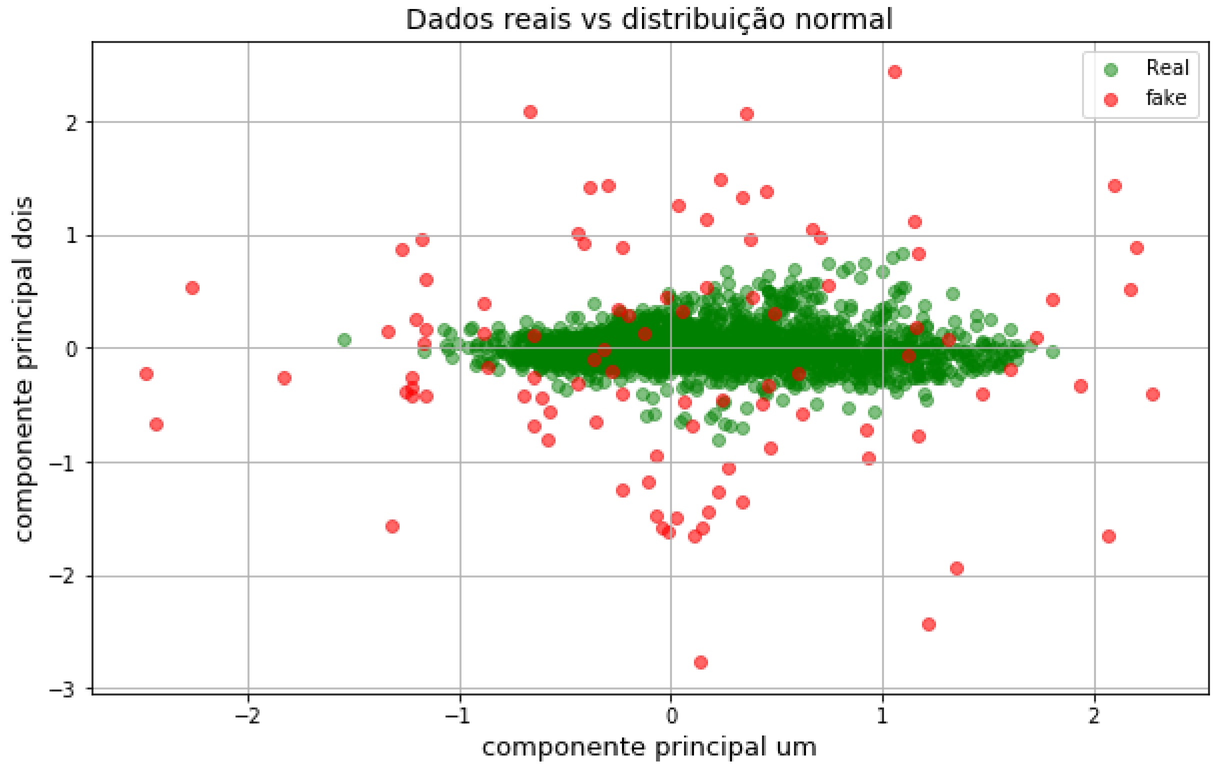
```
----------------------------- Genomas representados na forma de uma matriz -------------------------
----
```



Dados fake

- Os dados que vão ser passados pro gerador, são dados de uma distribuição normal.
- O gráfico abaixo mostra os dados reais e os dados da distribuição normal os quais são para trenar a rede geradora.

In [19]:
```
pca2 = PCA(n_components=2)
pca2.fit(X_train)
X_real = pca2.transform(X_train)
X_fake = pca2.transform(np.random.normal(0,1,(100,9)))
plt.figure(figsize=(10,6))
plt.grid()
plt.title("Dados reais vs distribuição normal", fontsize=14)
plt.xlabel("componente principal um", fontsize=13)
plt.ylabel("componente principal dois", fontsize=13)
plt.scatter(X_real[:,0], X_real[:,1], label="Real", alpha=.5, color="green")
plt.scatter(X_fake[:,0], X_fake[:,1], label="fake", alpha=.6, color="red")
plt.legend();
```

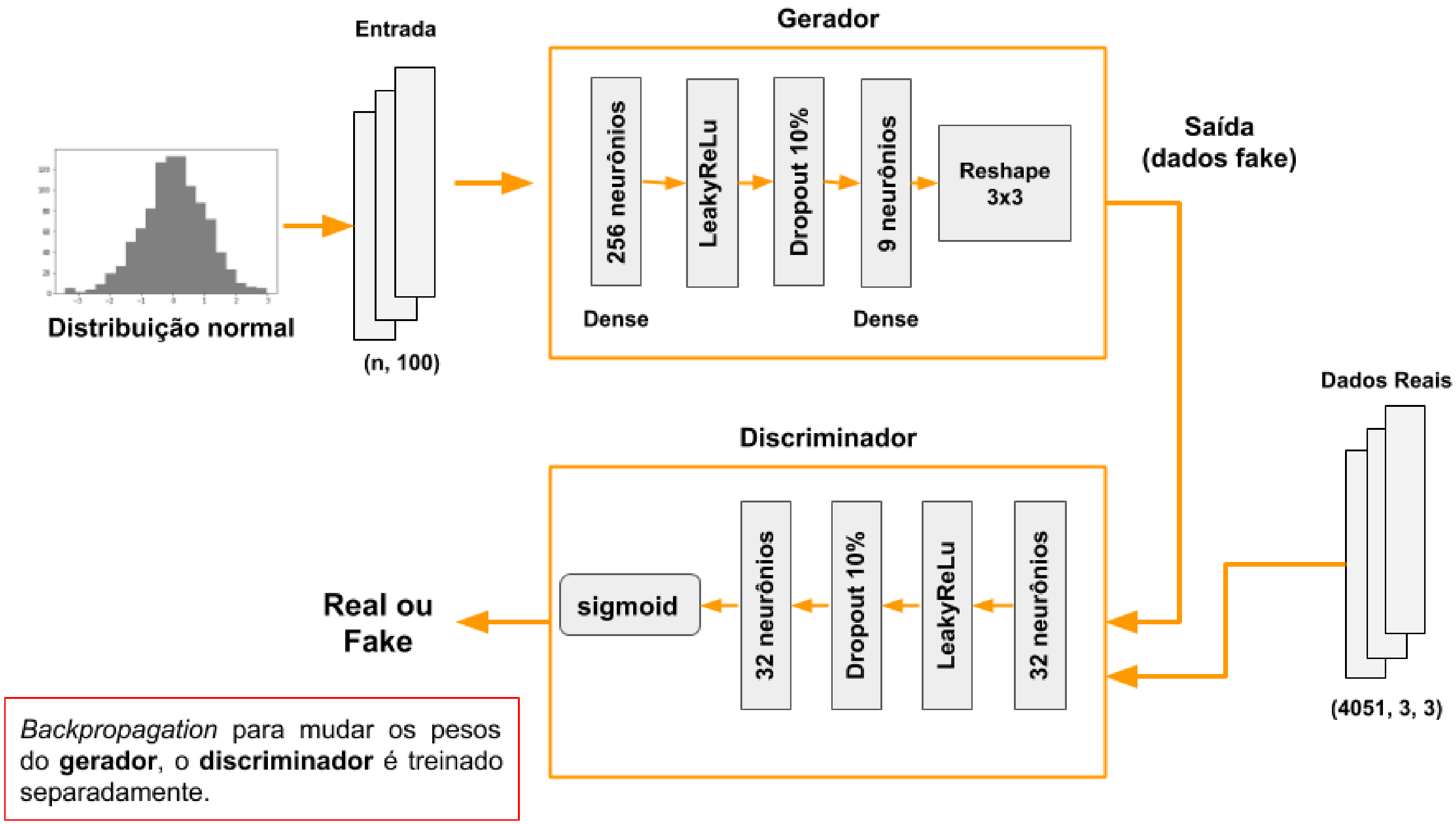


Construção da GAN

- A GAN conta com dois redes, uma rede geradora e uma discriminadora.

In [20]:
```
Image.open("./images/image1.png")
```

Out[20]:



In [21]:
```
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.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(256, input_shape=(self.noise_input,)))
        model.add(LeakyReLU(alpha=0.3))
        model.add(Dropout(.1))
        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(32, input_shape=self.dim))
        model.add(LeakyReLU(alpha=0.2)) #función rectificadora
        model.add(Dropout(.1))
```

```
        model.add(Dense(32))
        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 = []
        self.g_lpe = []
```

```
        self.d_lpe = []
        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))))
                #misturar os dados
                #x_combined_batch, y_combined_batch = shuffle(x_combined_batch, y_combined_batch
                #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%len(batches))==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])
```

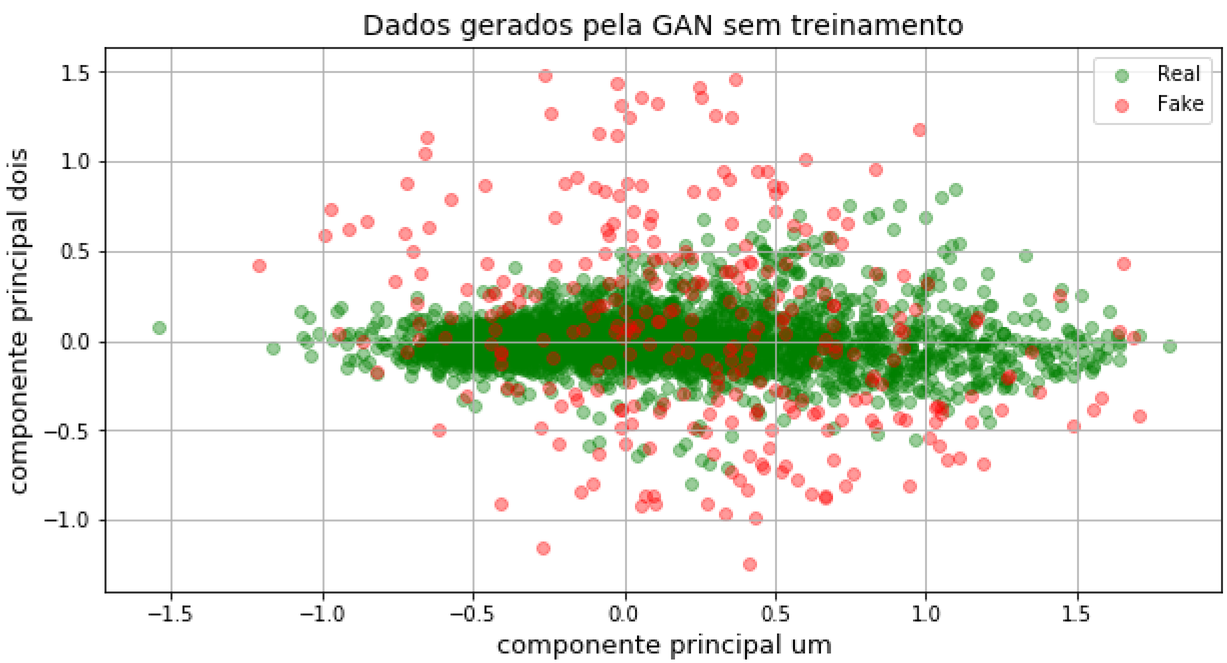
In [22]:

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

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

In [23]:
```
num_imgs = 300
fakes = gan.G.predict(np.random.normal(0,1,(num_imgs,noise_input)))
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)
plt.figure(figsize=(10,5))
plt.title("Dados gerados pela GAN sem treinamento", 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();
```

```
imagens fake:  (300, 3, 3, 1)
re-dimesionalidade:  (300, 9)
```



In [24]:

```
print("----------------------Estrutura da rede generativa----------------------")
gan.G.summary()
```

```
----------------------Estrutura da rede generativa----------------------
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 256)               25856     
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 256)               0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 9)                 2313      
_________________________________________________________________
reshape_1 (Reshape)          (None, 3, 3, 1)           0         
=================================================================
Total params: 28,169
Trainable params: 28,169
Non-trainable params: 0
_________________________________________________________________
```

In [25]:
```
print("----------------------Estrutura da rede Discriminadora----------------------")
gan.D.summary()
```

```
----------------------Estrutura da rede Discriminadora----------------------
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_3 (Dense)              (None, 3, 3, 32)          64        
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 3, 3, 32)          0         
_________________________________________________________________
dropout_2 (Dropout)          (None, 3, 3, 32)          0         
_________________________________________________________________
dense_4 (Dense)              (None, 3, 3, 32)          1056      
_________________________________________________________________
flatten_1 (Flatten)          (None, 288)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 289       
=================================================================
Total params: 2,818
Trainable params: 1,409
Non-trainable params: 1,409
_________________________________________________________________

/home/ejrueda/anaconda3/lib/python3.6/site-packages/keras/engine/training.py:490: UserWarning: Discr
epancy between trainable weights and collected trainable weights, did you set `model.trainable` with
out calling `model.compile` after ?
  'Discrepancy between trainable weights and collected trainable'
```

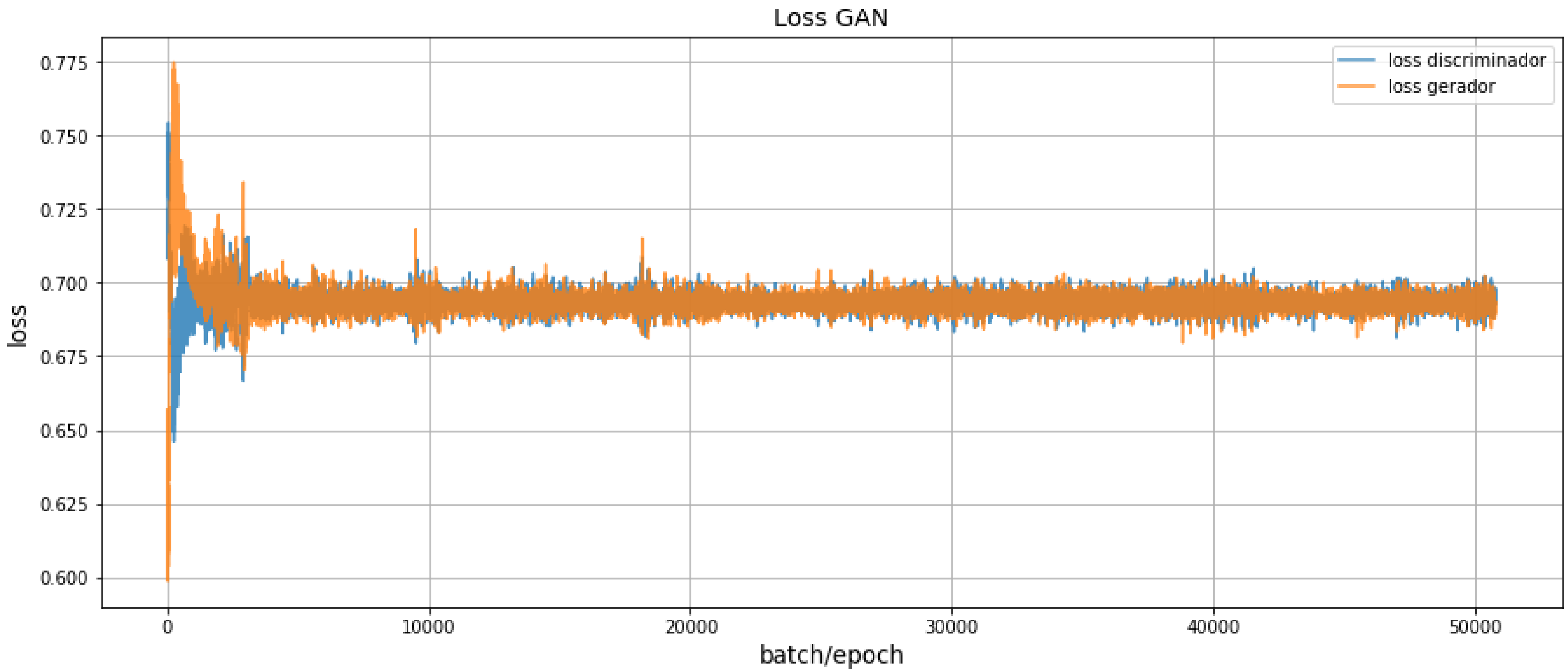
In [26]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [ ]:
```
t_i = time()
gan.train(X_img, epochs=400,batch_size=32)
t_f = time()
```

In [28]:
```
print("tempo de execução: ", (t_f-t_i)/60, "[min]")
```

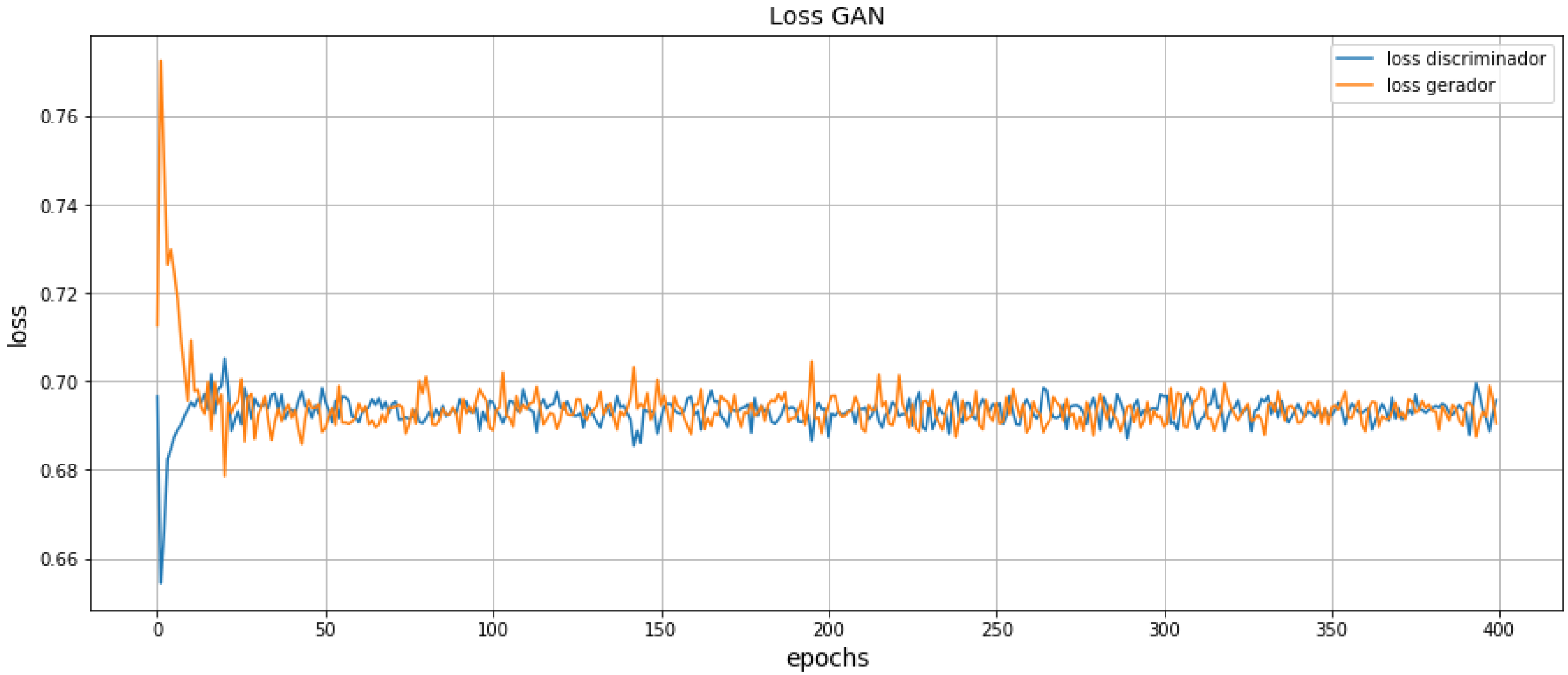
```
tempo de execução:  2.7460547208786013 [min]
```

In [29]:
```
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();
```



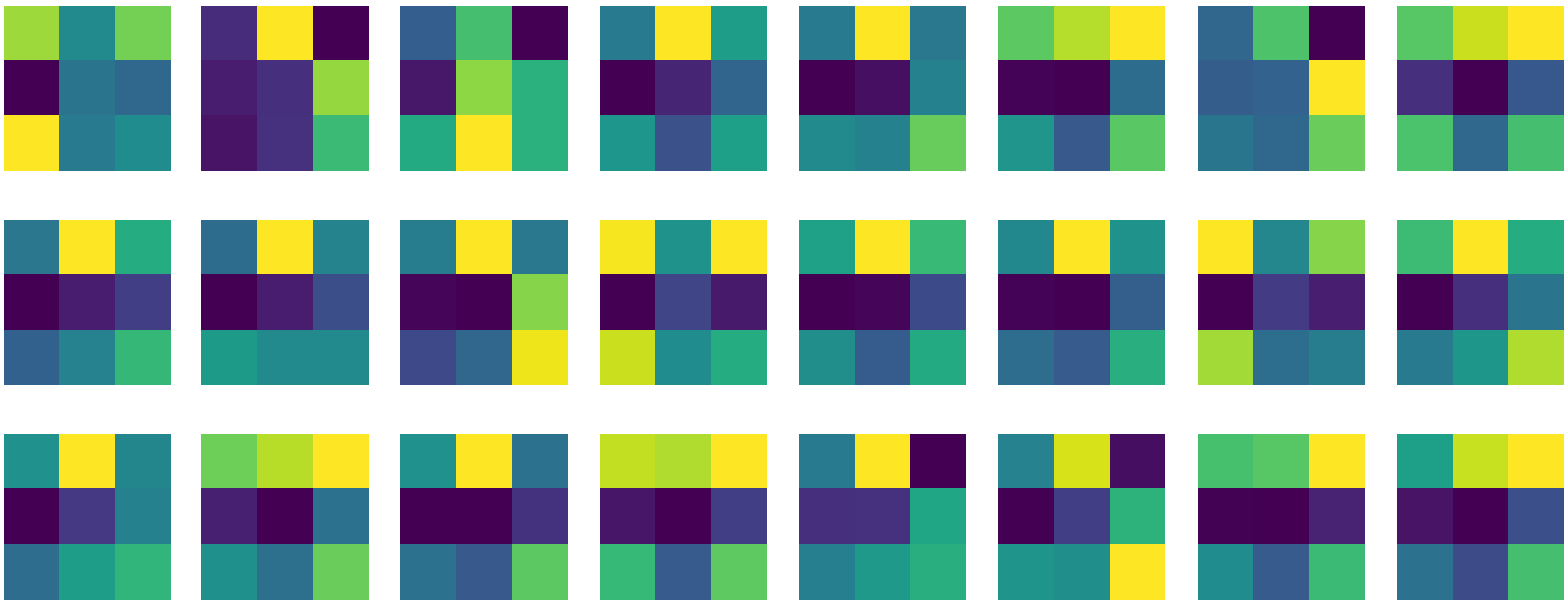
In [ ]:
```

```

In [30]:
```
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(len(gan.g_lpe)), 0.5*np.ones(len(gan.g_lpe)),
#          color="black", label="objetivo", linestyle='--')
plt.grid()
plt.legend();
```



In [ ]:
```

```

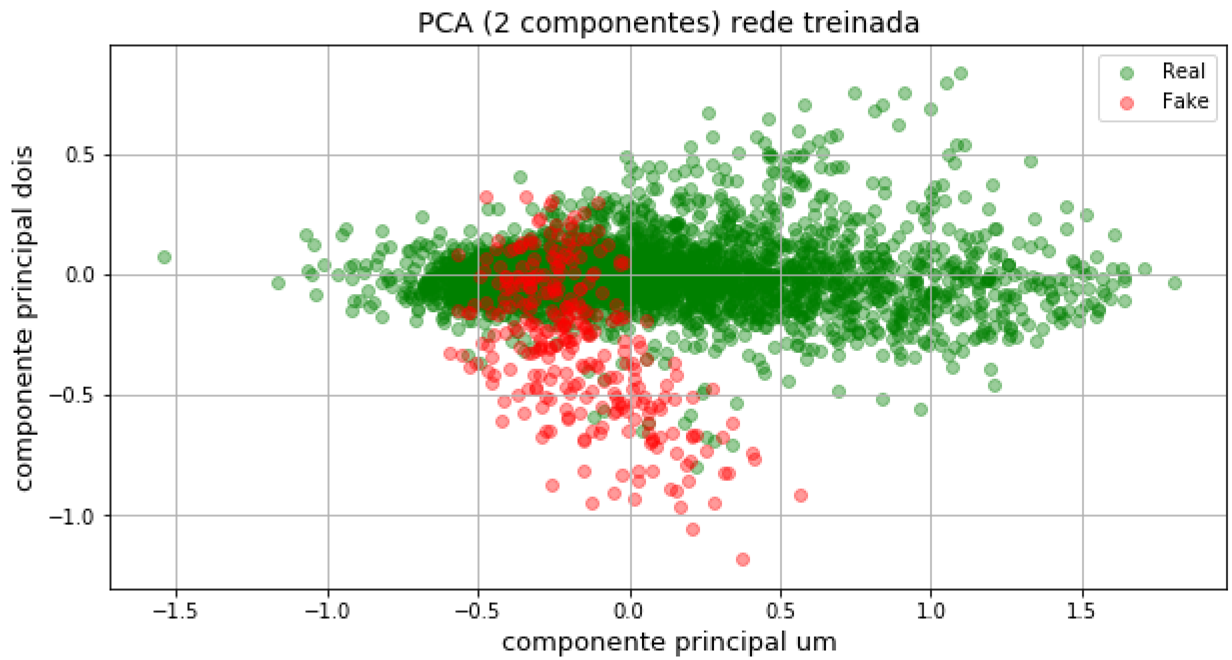
In [31]:
```
num_imgs = 24 #número de imágenes a mostrar aleatoriamente
img_pre = gan.G.predict(np.random.normal(0,1,(num_imgs, noise_input)))
fig = plt.figure(figsize=(15,6))
for i in range(num_imgs):
    ax=fig.add_subplot(3,8,i+1)
    img = img_pre[i]
    plt.imshow(img.reshape((3,3)))
    ax.axis("off")
plt.show()
```



In [32]:
```
num_imgs = 300
fakes = gan.G.predict(np.random.normal(0,1,(num_imgs,noise_input)))
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:  (300, 3, 3, 1)
re-dimesionalidade:  (300, 9)
```

In [33]:
```
plt.figure(figsize=(10,5))
plt.title("PCA (2 componentes) rede treinada", 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();
```



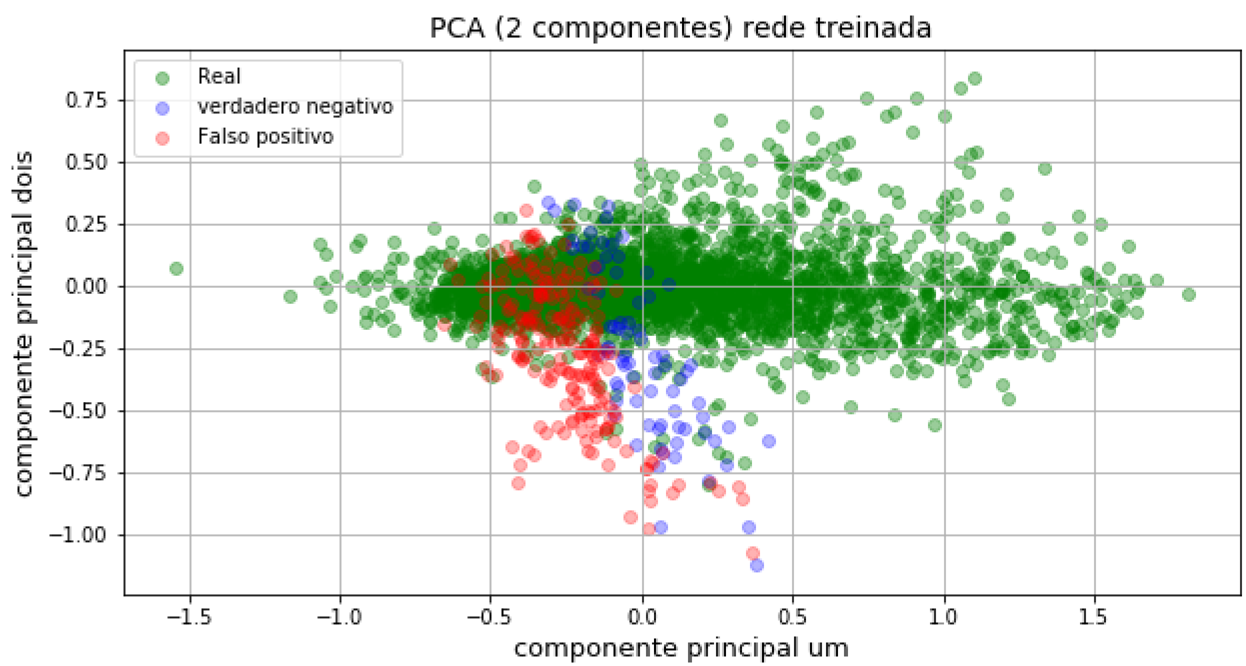
In [ ]:
```

```

In [34]:

```
num_imgs = 300
fakes = gan.G.predict(np.random.normal(0,1,(num_imgs,noise_input)))
print("Dados fake: ", fakes.shape)
y_predict = gan.D.predict_classes(fakes)
print("-------------- Discriminador --------------")
print("porcentagem de dados fake que o discriminador acredita reais: ", np.mean(y_predict==1))
print("porcentagem de dados fake que o discriminador acredita fakes: ", np.mean(y_predict==0))
print()
y_predict2 = gan.D.predict_classes(X_img)
print("-------------- Discriminador com dados reais --------------")
print("porcentagem de dados reais que o discriminador acredita reais: ", np.mean(y_predict2==1))
print("porcentagem de dados reais que o discriminador acredita fakes: ", np.mean(y_predict2==0))
```

```
Dados fake:  (300, 3, 3, 1)
-------------- Discriminador --------------
porcentagem de dados fake que o discriminador acredita reais:  0.74
porcentagem de dados fake que o discriminador acredita fakes:  0.26

-------------- Discriminador com dados reais --------------
porcentagem de dados reais que o discriminador acredita reais:  0.44235991113305356
porcentagem de dados reais que o discriminador acredita fakes:  0.5576400888669464
```

In [35]:
```
fakes = fakes.reshape(num_imgs,9)

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)
X_fp = X_fake[np.where(y_predict==1)[0]]
X_vn = X_fake[np.where(y_predict==0)[0]]

plt.figure(figsize=(10,5))
plt.title("PCA (2 componentes) rede treinada", 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_vn[:,0], X_vn[:,1], color="blue", alpha=.3, label="verdadero negativo")
plt.scatter(X_fp[:,0], X_fp[:,1], color="red", alpha=.3, label="Falso positivo")
plt.legend();
```



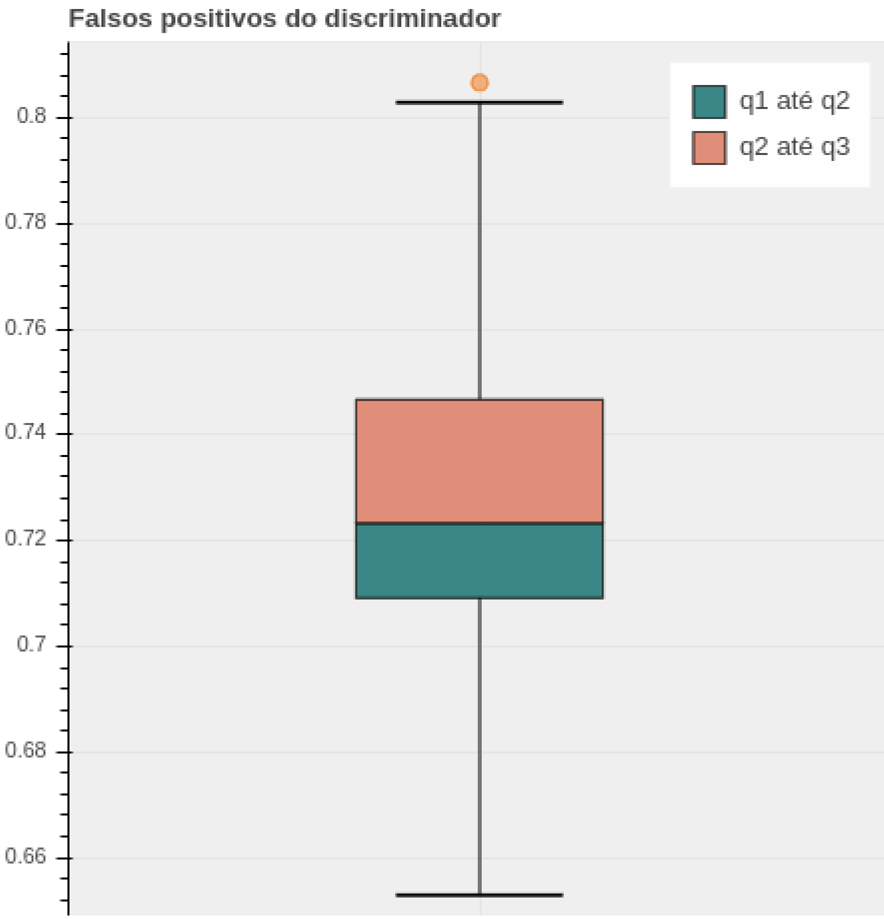
In [36]:
```
bp_fp = []
bp_vn = []
for i in range(300):
    fakes = gan.G.predict(np.random.normal(0,1,(num_imgs,noise_input)))
    y_predict = gan.D.predict_classes(fakes)
    bp_fp.append(np.mean(y_predict==1))
    bp_vn.append(np.mean(y_predict==0))
```

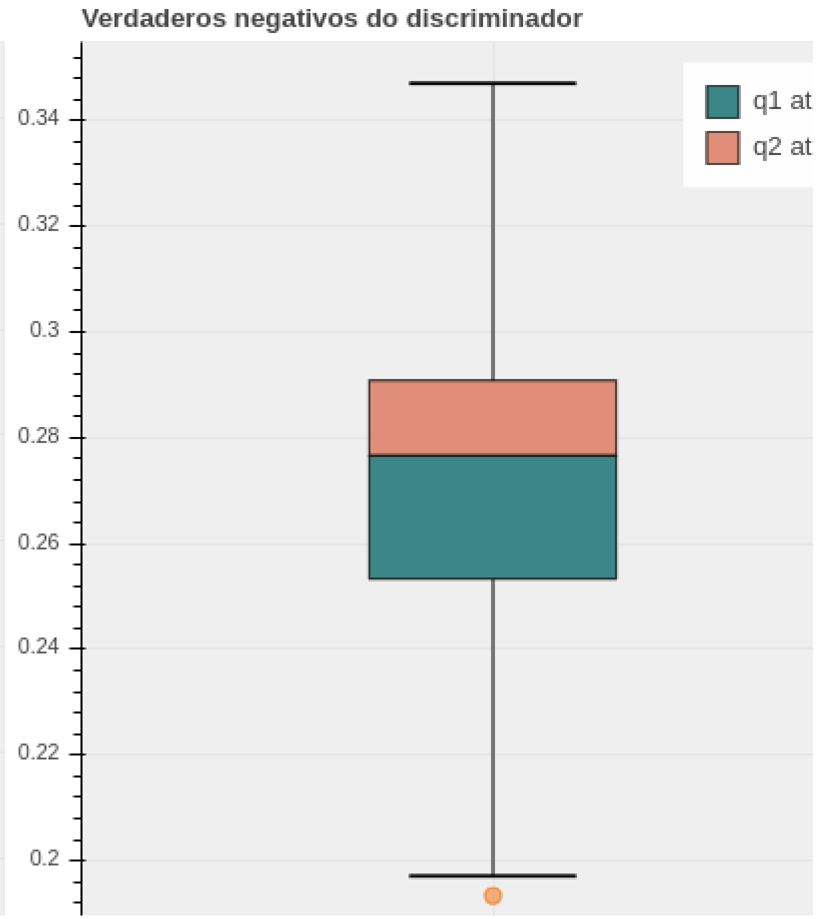
In [37]:

```
from bokeh.plotting import figure, show, output_file, output_notebook
from bokeh.layouts import row
from bokeh.models import ColumnDataSource

output_notebook()
x_ticks = ["falsos_positivos"]
p1 = figure(tools="", background_fill_color="#efefef", toolbar_location=None, x_range=x_ticks,
           title="Falsos positivos do discriminador", width=450, height=500)
q1 = np.quantile(bp_fp, q=0.25)
q2 = np.quantile(bp_fp, q=0.5)
q3 = np.quantile(bp_fp, q=0.75)
iqr = q3 - q1
upper = q3 + 1.5*iqr
lower = q1 - 1.5*iqr
outliers = np.array(bp_fp)[(bp_fp>upper) + (bp_fp<lower)]
source1 = ColumnDataSource(dict(x=x_ticks, upper=[upper], lower=[lower], q1=[q1], q2=[q2], q3=[q3
source2 = ColumnDataSource(dict(x=x_ticks*len(outliers), y=outliers))
#Para graficar las lineas superiores del boxplot
p1.rect("x","upper",0.2,0.0003, line_color="black", fill_color="black", source=source1)
#Para graficar las lineas inferiores del boxplot
p1.rect("x", "lower", 0.2, 0.0003, line_color="black", fill_color="black", source=source1)
#Para graficar los segmentos del boxplot
p1.segment("x", "lower", "x", "q1", line_color="black", source=source1)
p1.segment("x", "upper", "x", "q3", line_color="black", source=source1)
#Para graficar las barras
p1.vbar("x", 0.3, "q1", "q2", fill_color="#3B8686", line_color="black", source=source1, legend="c
p1.vbar("x", 0.3, "q2", "q3", fill_color="#E08E79", line_color="black", source=source1, legend="c
p1.circle("x","y", size=8, color="#F38630", fill_alpha=0.6, source=source2)

x_ticks2 = ["verdaderos negativos"]
p2 = figure(tools="", background_fill_color="#efefef", toolbar_location=None, x_range=x_ticks2,
           title="Verdaderos negativos do discriminador", width=450, height=500)
p2_q1 = np.quantile(bp_vn, q=0.25)
p2_q2 = np.quantile(bp_vn, q=0.5)
p2_q3 = np.quantile(bp_vn, q=0.75)
p2_iqr = p2_q3 - p2_q1
p2_upper = p2_q3 + 1.5*iqr
p2_lower = p2_q1 - 1.5*iqr
outliers2 = np.array(bp_vn)[(bp_vn>p2_upper) + (bp_vn<p2_lower)]
source3 = ColumnDataSource(dict(x=x_ticks2, upper=[p2_upper], lower=[p2_lower],
                                q1=[p2_q1], q2=[p2_q2], q3=[p2_q3]))
source4 = ColumnDataSource(dict(x=x_ticks2*len(outliers2), y=outliers2))
```

```
#Para graficar las lineas superiores del boxplot
p2.rect("x","upper",0.2,0.0003, line_color="black", fill_color="black", source=source3)
#Para graficar las lineas inferiores del boxplot
p2.rect("x", "lower", 0.2, 0.0003, line_color="black", fill_color="black", source=source3)
#Para graficar los segmentos del boxplot
p2.segment("x", "lower", "x", "q1", line_color="black", source=source3)
p2.segment("x", "upper", "x", "q3", line_color="black", source=source3)
#Para graficar las barras
p2.vbar("x", 0.3, "q1", "q2", fill_color="#3B8686", line_color="black", source=source3, legend="c
p2.vbar("x", 0.3, "q2", "q3", fill_color="#E08E79", line_color="black", source=source3, legend="c
p2.circle("x","y", size=8, color="#F38630", fill_alpha=0.6, source=source4)

show(row([p1,p2]))
```

(https://bokeh.pydata.org) BokehJS 1.1.0 successfully loaded.





In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [187]:
```
#gan.D.save_weights("D_weights.h5")
#gan.G.save_weights("G_weights.h5")
#gan.stacked_generator_discriminator.save_weights("stacked_weights.h5")
```

In [ ]:
```

```