INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO, ITAM
Curso: Visión por Computadora

Tarea 6

Pablo Barranco Soto 151528
Sebastián Martínez Santos 176357

1. GAN

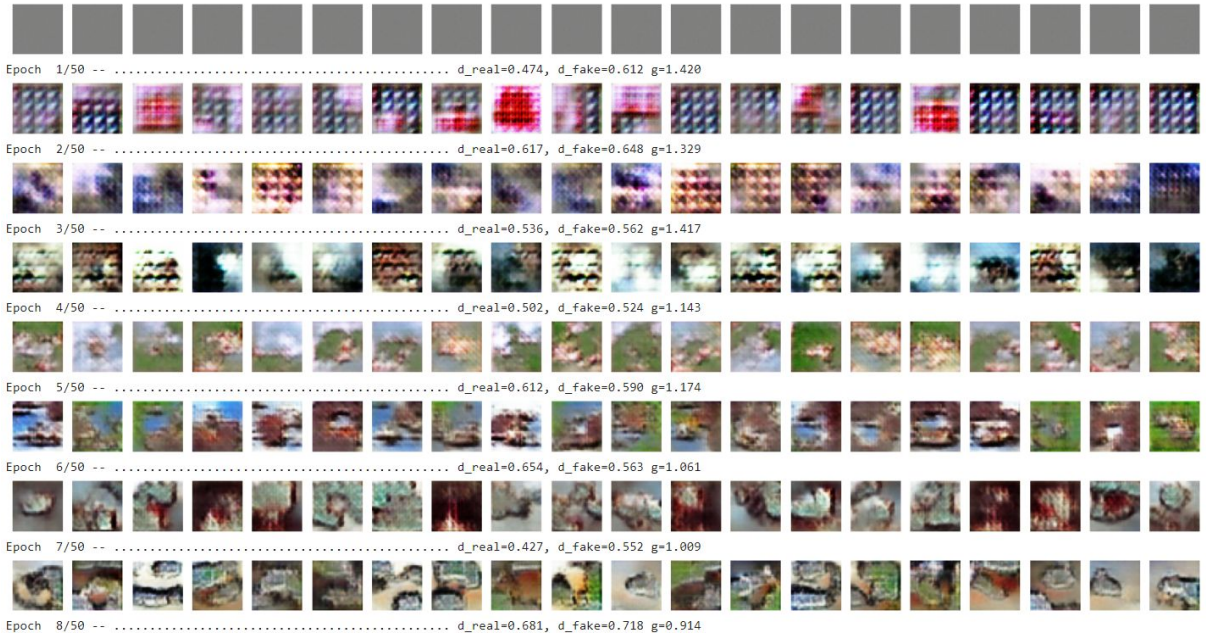
Para mejorar el GAN optamos por aumentar el número de capas tanto del discriminador como del generador. Con este cambio notamos cierta mejoría en las imágenes resultantes sin que se sacrificara mucho en complejidad computacional.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(32,32,3)):
  model = Sequential()

  # normal
  model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))

  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))

  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))

  # downsample
  model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))

  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))

  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

```python
# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 4x4 image
  n_nodes = 256 * 4 * 4
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((4, 4, 256)))
  # upsample to 8x8
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 16x16
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 32x32
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # output layer
  model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
  return model
```

Realizamos 50 épocas exitosamente.

Primeras:



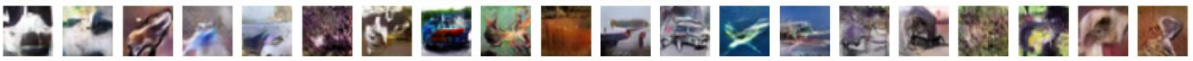
Últimas:

Epoch 43/50 -- ..... d_real=0.629, d_fake=0.602 g=0.966

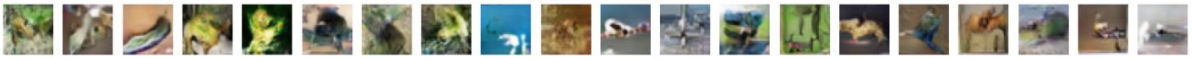
Epoch 44/50 -- ..... d_real=0.770, d_fake=0.477 g=0.999

Epoch 45/50 -- ..... d_real=0.685, d_fake=0.643 g=0.912

Epoch 46/50 -- ..... d_real=0.684, d_fake=0.705 g=0.945

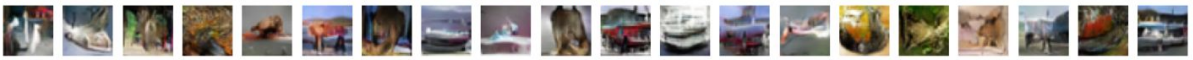


Epoch 47/50 -- ..... d_real=0.569, d_fake=0.603 g=1.007

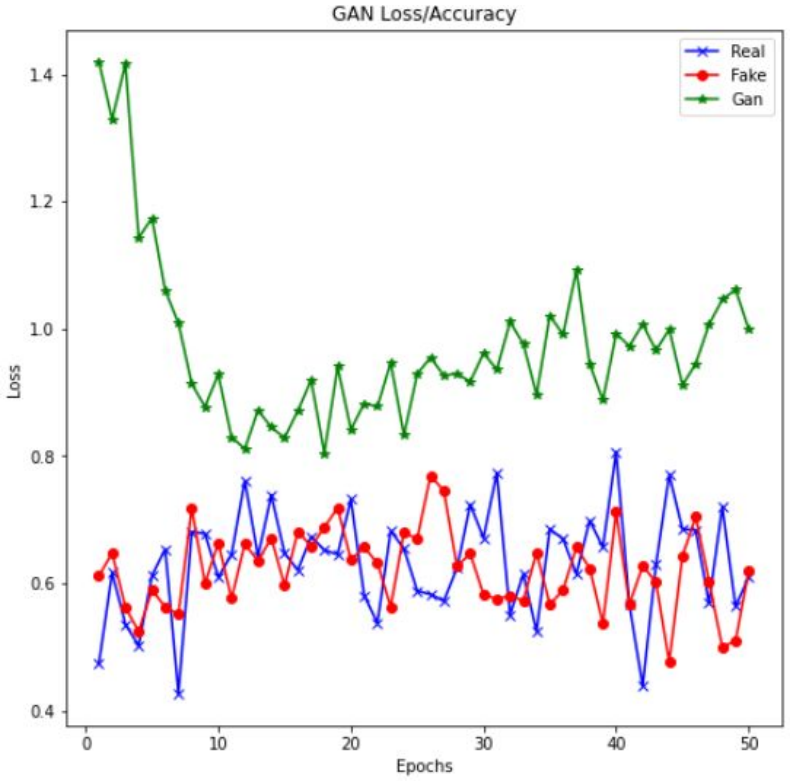


Epoch 48/50 -- ..... d_real=0.721, d_fake=0.500 g=1.046

Epoch 49/50 -- ..... d_real=0.564, d_fake=0.510 g=1.062



Epoch 50/50 -- ..... d_real=0.610, d_fake=0.620 g=0.999



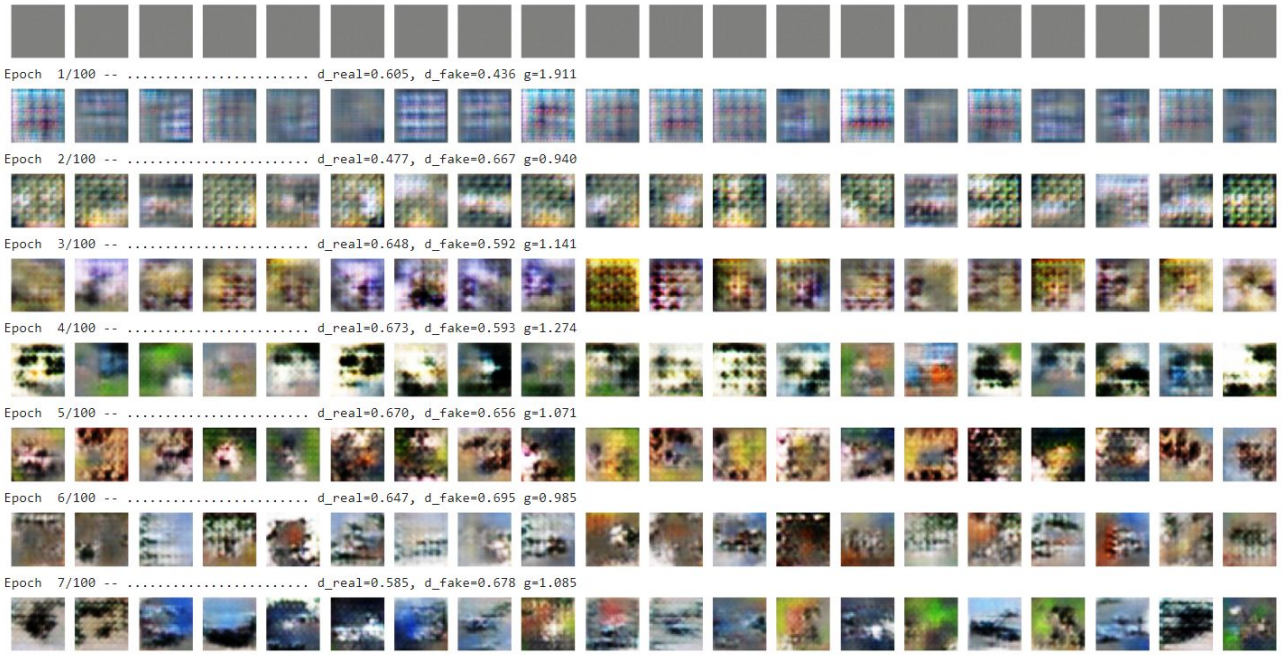
2. CGAN

```
# define the standalone generator model
def define_generator(latent_dim,n_classes = 10):

  #label input
  in_label = Input(shape = (1,))

  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)

  # linear multiplication
  n_nodes = 4*4
  li = Dense(n_nodes)(li)
  # reshape to additional channel
  li = Reshape((4, 4, 1))(li)
  # image generator input
  in_lat = Input(shape=(latent_dim,))

  # foundation for 4x4 image
  n_nodes = 256 * 4 * 4
  gen = Dense(n_nodes)(in_lat)
  gen = LeakyReLU(alpha=0.2)(gen)
  gen = Reshape((4, 4, 256))(gen)

  # merge image gen and label input
  merge = Concatenate()([gen, li])

  # upsample to 8x8
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(merge)
  gen = LeakyReLU(alpha=0.2)(gen)

  # upsample to 16x16
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
  gen = LeakyReLU(alpha=0.2)(gen)

  # upsample to 32x32
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
  gen = LeakyReLU(alpha=0.2)(gen)

  # output layer
  out_layer = Conv2D(3, (3,3), activation='tanh', padding='same')(gen)

  # define model
  model = Model([in_lat,in_label],out_layer)
  return model
```

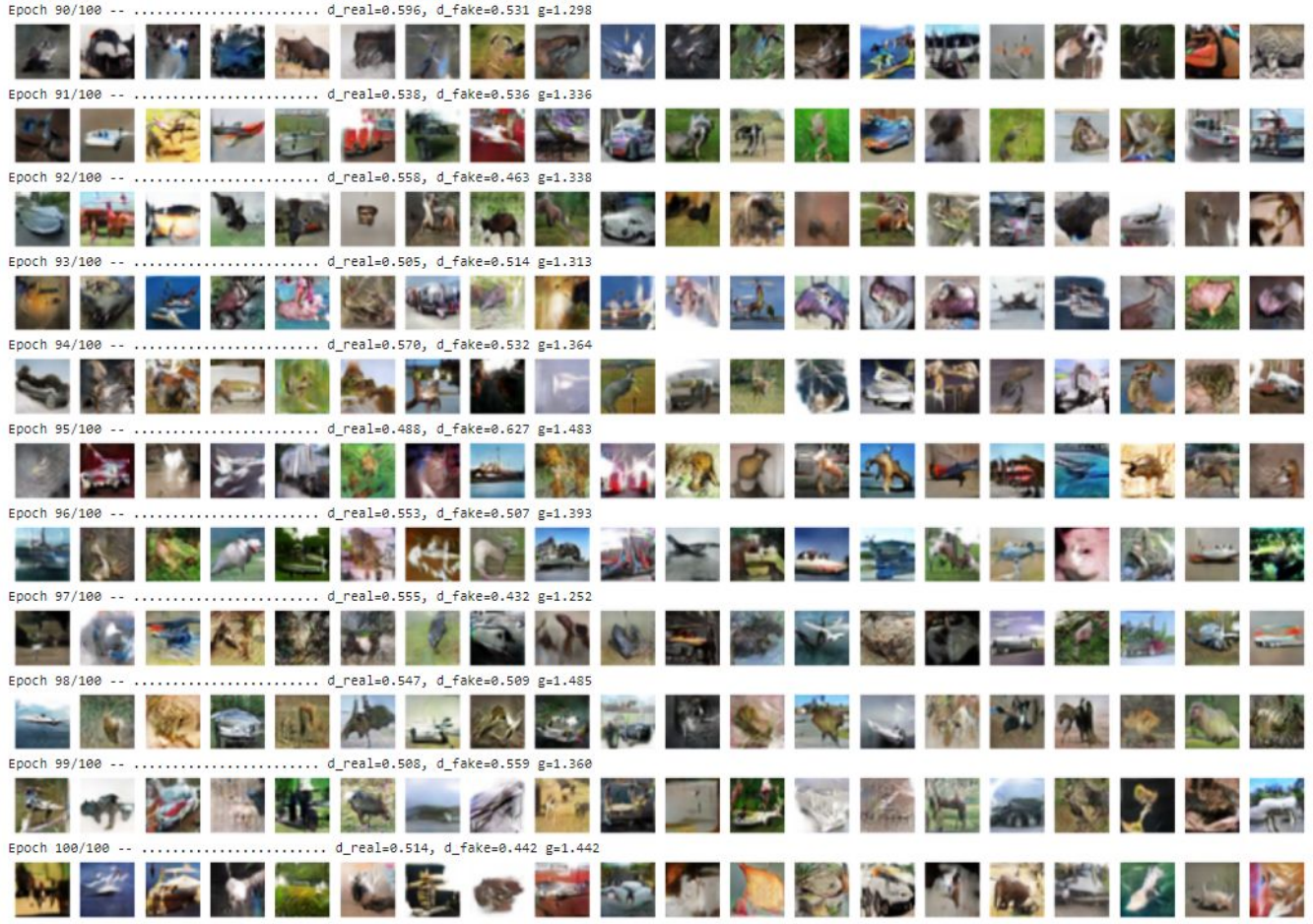
```python
# Redefinimos el discriminador
def define_discriminator(in_shape=(32,32,3), n_classes = 10):

  # label input
  in_label = Input(shape=(1,))

  # embedding for categorical input
  li = Embedding(n_classes,50)(in_label)

  # Scale up to image dinesions with linear activation
  n_nodes = in_shape[0]*in_shape[1]
  li = Dense(n_nodes)(li)

  # reshape to additional channel
  li = Reshape((in_shape[0], in_shape[1],1))(li)

  #image input
  in_image = Input(shape = in_shape)

  # concat label as a channel
  merge = Concatenate()([in_image,li])

  # normal
  fe = Conv2D(64, (3,3), padding='same')(merge)
  fe = LeakyReLU(alpha=0.2)(fe)

  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)

  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)

  # downsample
  fe = Conv2D(256, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)

  # classifier
  fe = Flatten()(fe)
  fe = Dropout(0.4)(fe)
  out_layer = Dense(1, activation='sigmoid')(fe)

  # compile model
  model = Model([in_image,in_label], out_layer)

  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

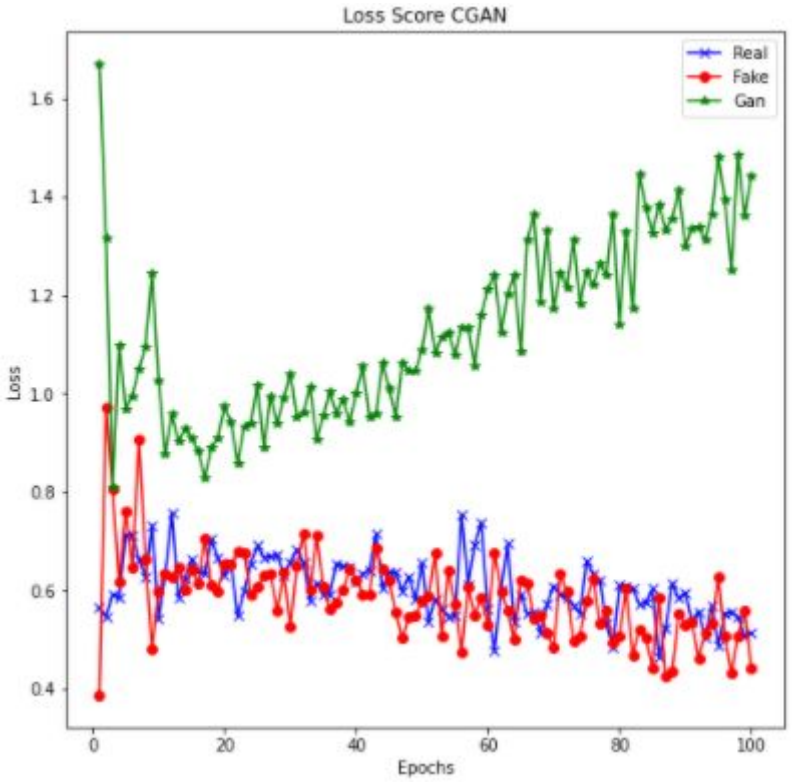
Realizamos 100 épocas exitosamente.

Primeras épocas:



Últimas épocas:





Podemos observar que las imágenes resultantes después de 50 épocas se parecen bastante a las reales. Dependiendo de la categoría es que vemos que hay un mejor desempeño. Por ejemplo, para la creación de automóviles esta DCGAN es bastante buena. Pues obtuvimos resultados como los siguientes en las últimas épocas:

3. WGAN

Para implementar el WGAN tomamos el modelo GAN y lo adaptamos para que usara la función de pérdida Wasserstein seguimos los siguientes pasos:

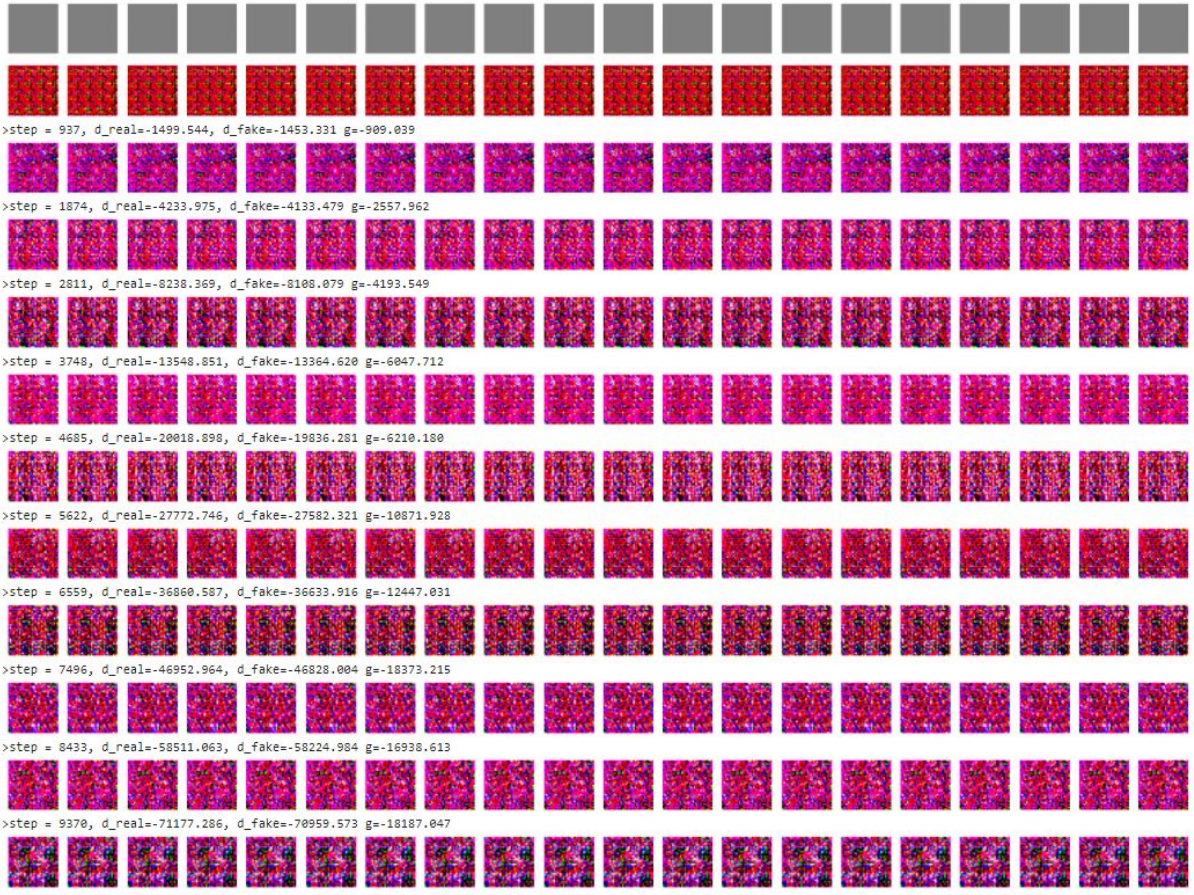
- Se cambió función de activación de la capa final del discriminador por una funciòn lineal
- Las etiquetas pasaron a ser -1 para las imágenes falsas y 1 para las reales.
- La función de pérdida del discriminador está dada por la función Wasserstein para el modelo del generador y del discriminador.
- El optimizador es RMSprop con learning rate de 0.0005.

- Actualizamos el discriminador 5 veces por cada vez que lo hacemos para el generador.

Como podemos observar en las siguientes imágenes se puede notar que aunque existe un cambio constante en las imágenes falsas, el resultado está muy lejos de parecerse a una imagen real. La función de pérdida de Wasserstein tiene la ventaja de converger, sin embargo no tuvimos tiempo o capacidad computacional suficiente para comprobarlo con este experimento.

Realizamos 50 épocas exitosamente.

Primeras épocas:



Últimas épocas: