Practica Deep Learning Architectures

Luis Osvaldo Cerna Copado | 18 may 2024

luis.cerna@cimat.mx

Diseño del Modelo

El autoencoder variacional es una variación del autoencoder tradicional que tiene el propópsito de mejorar la distribución del mapeo en el espacio latente. Puesto que el autoencoder tradicional genera huecos en el espacio latente, el autoencoder variacional emplea tecnicas estocasticas para corregir esto.

En la primera mitad del VAE, se mapea del input a una distribución probabilistica, generalmente una distribución gaussiana, en lugar de asignarle un punto en el espacio latente, se tiene un punto central (la media de la distribución) y un circulo de radio la varianza de la distribución. Además, debemos añadir una función de costo extra, conocida como divergencia KL, la cual nos ayuda a obtener distribuciones similares a la guassiana. Veremos que una de las ventajas de estos cambios es la generación de imagenes de mucho mejor calidad.

Encoder

Como primer paso necesitamos definir la parte de la codificación, la cual tiene 3 capas convolucionales que reducen el tamaño de la imagen a la mitad con un kernel de tamaño 4x4. En todas estas capas contamos con un función de activación relu, pues es la que mejor funciona en las capas intermedias. Una vez que las imagenes se redución hasta 3 veces, pasamos de tener 2 dimenciones a tener una sola y poder trabajar más fácilmente los valores de la distribución. Así, esta última capa densa con activación sigmoide nos genera dos vectores del tamaño del espacio latente, uno conteniendo los valores de la media y el otro los valores de la varianza.

```
x = layers.Conv2D(32, (4, 4), activation="relu")(encoder_input)
x = layers.Conv2D(64, (4, 4), activation="relu")(x)
x = layers.Conv2D(128, (4, 4), activation="relu")(x)
shape_before_flattening = K.int_shape(x)[1:]
x = layers.Flatten()(x)

z_mean = layers.Dense(embed_dim, name="z_mean")(x)
z_log_var = layers.Dense(embed_dim, name="z_log_var")(x)

z = Sampling()([z_mean, z_log_var])
```

La útlima parte del encoder requiere de una función nueva que nos presenta la módificación variacional de esta red neuronal. La función Sampling nos ayuda a pasar de nuestras distribuciones a un punto particular en el espacio latente. Tomamos una muestra de cada una de nuestras distribuciones con ayuda de una normal. Se trabaja con la log_var para poder emplear valores negativos.

```
class Sampling(layers.Layer):
  def call(self, inputs):
    z_mean, z_log_var = inputs
    batch = tf.shape(z_mean)[0]
    dim = tf.shape(z_mean)[1]
    epsilon = K.random_normal(shape=(batch, dim))
    return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

Decoder

Una vez que tenemos nuestro espacio latente como muestras de distribuciones normales, el decodificador lo único que hace es retroceder sobre sus mismos pasos. Una capa densa para obtener la dimención original que guardamos en el encoder, 3 capas convolucionales transpuestas con activaciones relu a excepción de la última que al igual que el encoder, cuenta con una activación sigmoide. Esta función de activación nos ayuda a decidir de forma contundente los valores que se asignan en las salidas.

```
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input)
x = layers.Reshape(shape_before_flattening)(x)
x = layers.Conv2DTranspose(128, (4, 4), activation="relu")(x)
x = layers.Conv2DTranspose(64, (4, 4), activation="relu")(x)
x = layers.Conv2DTranspose(32, (4, 4), activation="relu")(x)
decoder_output = layers.Conv2D(3,(4, 4),activation="sigmoid",)(x)
```

VAE

Para poder unir todos estos elementos creamos una clase VAE. El uso de una clase nos va a permitir generar funciones de entrenamiento y prueba personalizadas para poder añadir la función de costo extra que necesitamos ahora que queremos corregir no solo el resultado de nuestros datos si no el de nuestras distribuciones.

```
class VAE(models.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        ...
        def call(self, inputs):
            z_mean, z_log_var, z = encoder(inputs)
            reconstruction = decoder(z)
            return z_mean, z_log_var, reconstruction

        def train_step(self, data):
            z_mean, z_log_var, reconstruction = self(data)
            beta = 500
            reconstruction_loss = tf.reduce_mean(
                beta* losses.binary_crossentropy(...)
            )
```

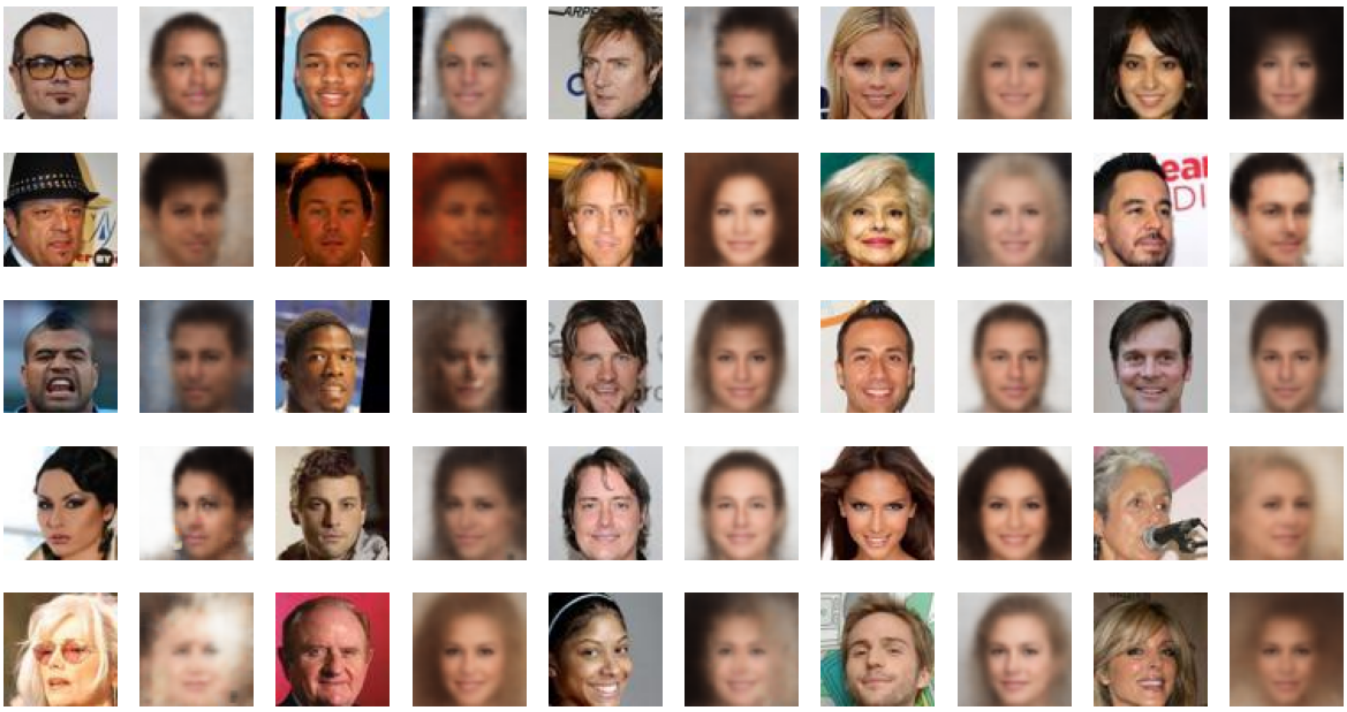
```
            kl_loss = tf.reduce_mean(
                tf.reduce_sum(
                    -0.5*(1+z_log_var-square(z_mean)-exp(z_log_var))))
            total_loss = reconstruction_loss + kl_loss
        ...

        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
```

Se optó por el optimizador Adam con learning rate de 0.001, el cual de muy buenos resultados en este caso. Además ya que estamos usando la función sigmoide en las capas de salida, empleamos la función de pedida binary cross entropy.

Resultados

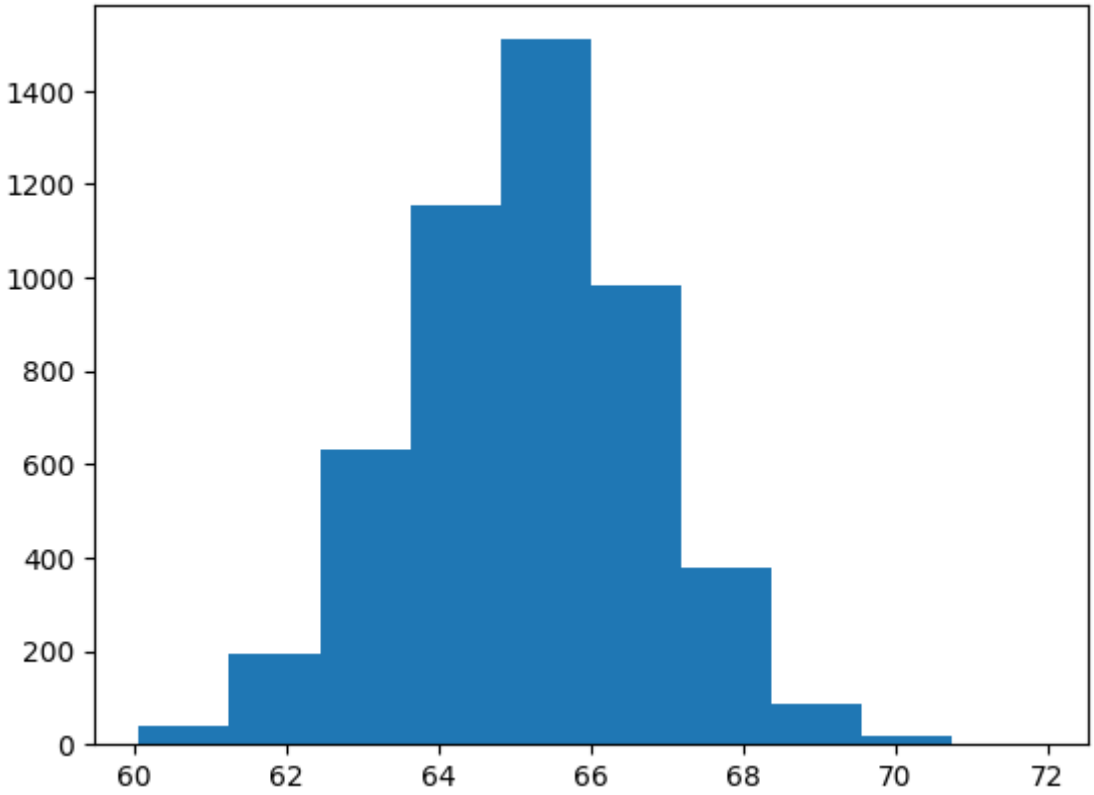
En esta ocación empleamos el conjunto de datos CelebA, que contiene más de 200,000 imagenes de rostros de famosos junto con una lista de los atributos fisicos que presentan cada uno de ellos. Esta lista sera útil en una aplicación extra que veremos al final de la sección de resultados. Puesto que el tamaño del data set era demasiado grande, se optó por un conjunto de datos reducida con un total de 50,000 imagenes de tamaño 64x64 las cuales funcionaron de forma muy similar a los resultados vistos con el total de las imagenes de mayor calidad.

Después de entrenar la red neuronal por 250 epocas con una dimensión de 100 en el espacio latente obtenemos los siguientes resultados. Se obtuvieron perdidas de 11.79 en la divergencia KL y 280 de binary cross entropy. La reconstrucción de las imagenes.



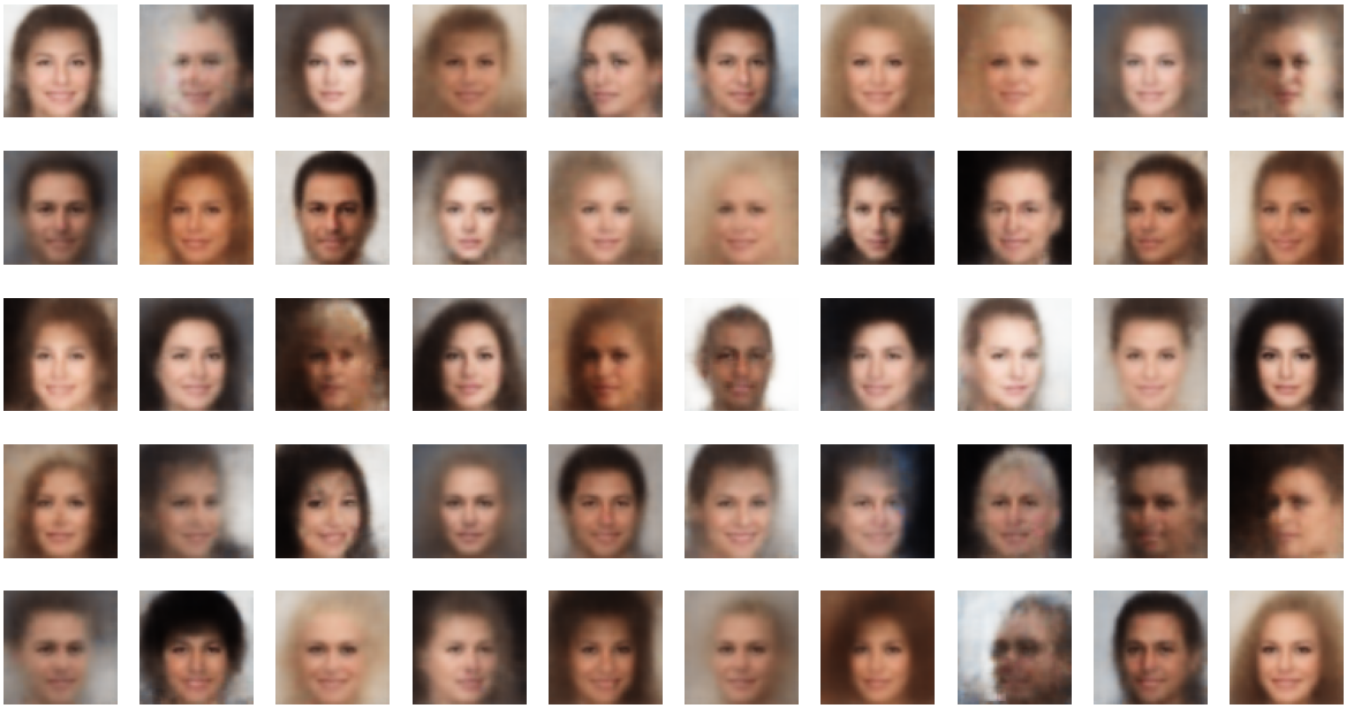
Encontramos que la mayoria de los resultados son muy buenos. Algunos de los errores en la reconstrucción se presentan cuando hay objetos obstruyendo el rostro, con sombreros, cuando el color del fondo es muy similar al color del cabello o cuando el rostro esta rotado. Sin embargo, la red cumple casi en su totalidad con el objetivo de recostruir un rostro similar al original.

Ahora, requerimos de una medida que nos ayude a definir que tan similares son las imagenes que se recrean en comparación con las originales. La medida que escogimos fue la de Peak signal-to-noise ratio. Por como esta definida, PSNR nos dice que mientras más alto sea el valor más parecidas son las imagenes. Mostramos un histograma de los valores obtenidos de PSNR para cada uno de las imagenes reconstruidas en el conjunto de prueba.



Como podemos ver, todos los valores se encuentran entre 60 y 70, los cuales son valores dentro del rango adecuado para esta escala de comparación de imagenes. También es muy destacable como no encontramos ningún valor atípico, lo que nos da una pista de que el algoritmo es robusto.

Finalmente, para generar imagenes de rostros completamente nuevos debemos obtener muestras aleatorias de nuestros espacio latente y pasarlas directamente por el decodificador. Los resultados son los siguientes.



Podemos ver que en su gran mayoria los resultados presentan rostros coherentes, aún que el cabello largo o rubio muchas veces le cuesta generarlo bien.

Como una aplicación extra y haciendo uso de la lista de atributos que nos proporciona la red neuronal, podemos cambiar gradualmente el color del cabello de una imagen en particular. Tomamos una imagen de un rostro con cabello negro y lo cambiaremos gradualemente a rubio con ayuda de nuestra red neuronal previamente entrenada.

Para lograr esto, definimos una función que nos ayuda a encontrar la dirección del vector en el espacio latente que nos lleva desde cualquier imagen al atributo que nos interesa. Para esto, debemos codificar cada una de las imagenes que presentan el atributo dado y promediamos el resultado de lo vectores resultantes, es decir la media y la varianza de las distribuciones. Repetimos el proceso con todas las imagenes que no cuentan con este atributo y restamos los vectores. De esta forma obtenemos la dirección que nos interesa

```
def get_feature_direction(VAE, list_attr, feature):
  with_feature = list_attr[list_attr[feature] == 1]
  wout_feature = list_attr[list_attr[feature] == -1]

  With_Feature = []
  for idx in with_feature_index_list[:cutoff]:
    With_Feature.append(VAE.encoder(image[idx]))

  Wout_Feature = []
  for idx in wout_feature_index_list[:cutoff]:
    Wout_Feature.append(VAE.encoder(image[idx]))

  With_Feature = np.mean(With_Feature, axis = 0)
  Wout_Feature = np.mean(Wout_Feature, axis = 0)

  return With_Feature - Wout_Feature
```

Una vez contamos con esta dirección basta con codificar la imagen de interes y decodificarla con una combinación linal del vector de interes y la dirección del atributo escalada para lograr el efecto gradual. El resultado es el siguiente.

Conclusión

De todo lo anterior podemos darnos cuenta de las impresionantes aplicaciones que se le pueden dar a las redes neuronales. La generación de nuevas imagenes y los buenos resultados que se pueden obtener con poca calidad en los datos nos da un panorama del potencial que tienen estos modelos en todo tipo de aplicaciones. En este caso no solo generamos rostros completamente nuevos, si no logramos modificar un rostro existente, lo que nos da una larga lista de aplicaciones nuevas.

En particular nuestro modelo logró resultados excepcionales aún cuando no se emplearon ni las imagenes en su calidad máxima, ni la totalidad de los datos. Conociendo bien los modelos y los procesos que cada uno conlleva nos ayuda a lograr mejores resultados. Un autoencoder convencional, por como se estructura, no hubiera logrado tan buenos resultados en esta tarea en especifico. Por otro lado, el autoencoder varacional corrige el problema del espacio latente casi a la perfección y nos da resultados impresionantes aún con pocos recursos de datos y hardwere.