Deep Learning.

Proyecto 1 - Autoencoders.

Nombre: Sergio Daniel Dueñas Godinez.

Expediente : 739300.

Profesor: Iván Reyes Amezcua.

Link Github: https://github.com/SergioDuenass/Camus-GPT

Objetivos:

```
Comprender los principios fundamentales de los autoencoders y su aplicación en deep learning generativo.
Implementar un autoencoder básico y variacionales para una tarea específica, como reducción de dimensión, denoising o gen
Analizar el rendimiento y las características de las representaciones aprendidas por los autoencoders.
```

Descripción

```
Deberán seleccionar un conjunto de datos adecuado para su proyecto, que puede ser de imágenes, texto o cualquier otro tip
```

Implementar un autoencoder, como un variacional (VAE) o un autoencoder convolucional, dependiendo de la naturaleza del co
El proyecto incluirá una fase de experimentación donde los deberán entrenar, ajustar y evaluar sus modelos.
Presentar sus resultados a través de un informe escrito y una presentación, discutir la implementación, los desafíos enco

Los datos fueron recopliados de los libros de Camus; 'El Extranjero' y 'La Plaga'

```
# Librerias a utilizar
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.losses import binary_crossentropy, mean_squared_error
from tensorflow.keras.layers import Input, Dense, Flatten, Reshape, Conv1D, Conv1DTranspose, Lambda

import numpy as np
import spacy
import re
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive"
```

Tokenización, embeddings y limpieza de corpus

```
def clean_corpus_from_file(file_path):
    cleaned_corpus = []

    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            # Leer el contenido del archivo
```

```
            corpus = file.readlines()

            for text in corpus:
                # Convertir a minúsculas
                text = text.lower()

                # Eliminar caracteres no alfabéticos y números
                text = re.sub(r'[^a-z\s]', '', text)

                # Eliminar espacios en blanco adicionales
                text = ' '.join(text.split())

                cleaned_corpus.append(text)

    except Exception as e:
        print(f"Error al leer el archivo: {e}")

    return cleaned_corpus



def clean_corpus(corpus):
    cleaned_corpus = []

    for text in corpus:
        # Convertir a minúsculas
        text = text.lower()

        # Eliminar caracteres no alfabéticos y números
        text = re.sub(r'[^a-z\s]', '', text)

        # Eliminar espacios en blanco adicionales
        text = ' '.join(text.split())

        cleaned_corpus.append(text)

    return cleaned_corpus
```

```
# Utilizo spacy para la tokenización
nlp = spacy.load("en_core_web_sm")

def tokenize_corpus(corpus):
    tokenized_corpus = []

    for text in corpus:
        # Tokenizar el texto usando spaCy
        tokens = [token.text for token in nlp(text)]
        tokenized_corpus.append(tokens)

    return tokenized_corpus

corpus_file_path = "/content/drive/MyDrive/ProyectoCamus/data/corpus.txt"
cleaned_corpus = clean_corpus_from_file(corpus_file_path)

tokenized_corpus = tokenize_corpus(cleaned_corpus)


# Construir un vocabulario
vocabulario = {token: idx for idx, token in enumerate(set(token for sublist in tokenized_corpus for token in s
vocab_size = len(vocabulario)

# Convertir tokens a índices
corpus_indices = [[vocabulario[token] for token in secuencia] for secuencia in tokenized_corpus]

# Padding de las secuencias
corpus_padded = pad_sequences(corpus_indices, padding='post', truncating='post')

# Crear el modelo de embedding
embedding_dim = 300  # ajusta según tus necesidades
embedding_model = Embedding(input_dim=vocab_size, output_dim=embedding_dim)

# Obtener vectores de embedding
embedded_sequence = embedding_model(corpus_padded)

# Ver la salida
print("Corpus Padded Shape:", corpus_padded.shape)
```

```
print("Corpus Padded Shape:", corpus_padded.shape)
print("Embedded Sequence Shape:", embedded_sequence.shape)
```

```
Corpus Padded Shape: (8289, 106)
Embedded Sequence Shape: (8289, 106, 300)
```

VAE

```
embedded_sequence.shape
```

```
TensorShape([8289, 106, 300])
```

```
input_shape = embedded_sequence.shape[1:]
batch_size = 128
latent_dim = 64
epochs = 30
```

Double-click (or enter) to edit

```
def sampling(args):
  z_mean, z_log_var = args

  dim = K.int_shape(z_mean)[1]

  # TODO: check dimensions
  epsilon = K.random_normal(shape = (K.shape(z_mean)[0], dim))

  return z_mean + K.exp(0.5 * z_log_var) * epsilon

# Ejemplo de capa de muestreo utilizando reparameterization trick
def sampling(args):
    z_mean, z_log_var = args
    batch = K.shape(z_mean)[0]
    dim = K.int_shape(z_mean)[1]
    epsilon = K.random_normal(shape=(batch, dim))
```

```
    return z_mean + K.exp(0.5 * z_log_var) * epsilon

# Ajustar las dimensiones de las secuencias de entrada
target_length_input = 106
corpus_padded_input = pad_sequences(corpus_indices, maxlen=target_length_input, padding='post', truncating='pc

# Convertir a embeddings
embedded_sequence_input = embedding_model(corpus_padded_input)

# Ajustar las dimensiones de las secuencias de salida
target_length_output = 106
corpus_padded_output = pad_sequences(corpus_indices, maxlen=target_length_output, padding='post', truncating='


# Convertir a embeddings
embedded_sequence_output = embedding_model(corpus_padded_output)
```

```
embedded_sequence_input.shape, embedded_sequence_output.shape
```

```
(TensorShape([8289, 106, 300]), TensorShape([8289, 106, 300]))
```

```
# Ajustar el modelo Encoder
input_shape_encoder = (target_length_output, embedding_dim)  # Ajusta según tus necesidades
latent_dim = 32  # Ajusta según tus necesidades

inputs_encoder = Input(shape=input_shape_encoder, name="encoder_input")
x_encoder = Conv1D(32, 3, activation="relu", strides=2, padding="same")(inputs_encoder)
x_encoder = Conv1D(64, 3, activation="relu", strides=2, padding="same")(x_encoder)

shape_before_flat_encoder = K.int_shape(x_encoder)
x_encoder = Flatten()(x_encoder)
x_encoder = Dense(256, activation="relu", kernel_initializer='he_normal')(x_encoder)

z_mean_encoder = Dense(latent_dim, name='z_mean')(x_encoder)
z_log_var_encoder = Dense(latent_dim, name='z_log_var')(x_encoder)
z = Lambda(sampling, output_shape=(latent_dim,))([z_mean_encoder, z_log_var_encoder])
```

```
z = Lambda(sampling, output_shape=(latent_dim,))([z_mean_encoder, z_log_var_encoder])

encoder = Model(inputs_encoder, [z_mean_encoder, z_log_var_encoder], name='encoder')

#
target_length_output = 106
embedding_dim = 300

# Definir el modelo Decoder
latent_inputs_decoder = Input(shape=(latent_dim,), name='z_sampling')
x_decoder = Dense(np.prod(shape_before_flat_encoder[1:]), activation="relu", kernel_initializer='he_normal')(la
x_decoder = Reshape(shape_before_flat_encoder[1:])(x_decoder)

# Usar Conv1DTranspose con padding='same' para ajustar la longitud de salida
x_decoder = Conv1DTranspose(64, 3, activation="relu", strides=2, padding="same")(x_decoder)
x_decoder = Conv1DTranspose(32, 3, activation="relu", strides=2, padding="same")(x_decoder)

# Ajustar manualmente la longitud de la salida a target_length_output
outputs_decoder = Conv1DTranspose(embedding_dim, 3, activation="linear", padding="same")(x_decoder)
outputs_decoder = outputs_decoder[:, :target_length_output, :]

# Definir el modelo Decoder
decoder = Model(latent_inputs_decoder, outputs_decoder, name='decoder')




# VAE
outputs_vae = decoder(encoder(inputs_encoder)[0])
vae = Model(inputs_encoder, outputs_vae, name='vae')


#

# reconstruction_loss = mean_squared_error(K.flatten(inputs), K.flatten(outputs) * input_shape[0] * input_shap
reconstruction_loss = mean_squared_error(K.flatten(inputs_encoder), K.flatten(outputs_vae))
```

```
kl_loss = 1 + z_mean_encoder - K.square(z_mean_encoder) - K.exp(z_log_var_encoder)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss)


vae.add_loss(vae_loss)
vae.compile(optimizer='adam')
vae.summary()
```

```
Model: "vae"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 encoder_input (InputLayer)     [(None, 106, 300)]   0           []

 encoder (Functional)           [(None, 32),         494112      ['encoder_input[0][0]']
                                 (None, 32)]

 decoder (Functional)           (None, 106, 300)     104652      ['encoder[0][0]']

 conv1d (Conv1D)                (None, 53, 32)       28832       ['encoder_input[0][0]']

 conv1d_1 (Conv1D)              (None, 27, 64)       6208        ['conv1d[0][0]']

 flatten (Flatten)              (None, 1728)         0           ['conv1d_1[0][0]']

 dense (Dense)                  (None, 256)          442624      ['flatten[0][0]']

 z_mean (Dense)                 (None, 32)           8224        ['dense[0][0]']

 tf.__operators__.add (TFOp     (None, 32)           0           ['z_mean[0][0]']
 Lambda)

 tf.math.square (TFOpLambda     (None, 32)           0           ['z_mean[0][0]']
 )

 z_log_var (Dense)              (None, 32)           8224        ['dense[0][0]']

 tf.reshape_1 (TFOpLambda)      (None,)              0           ['decoder[0][0]']
```

```
 tf.reshape (TFOpLambda)     (None,)                         0           ['encoder_input[0][0]']

 tf.math.subtract (TFOpLamb  (None, 32)                      0           ['tf.__operators__.add[0][0]',
 da)                                                                      'tf.math.square[0][0]']

 tf.math.exp (TFOpLambda)    (None, 32)                      0           ['z_log_var[0][0]']

 tf.convert_to_tensor (TFOp  (None,)                         0           ['tf.reshape_1[0][0]']
 Lambda)

 tf.cast (TFOpLambda)        (None,)                         0           ['tf.reshape[0][0]']

 tf.math.subtract_1 (TFOpLa  (None, 32)                      0           ['tf.math.subtract[0][0]',
 mbda)                                                                    'tf.math.exp[0][0]']

 tf.math.squared_difference  (None,)                         0           ['tf.convert_to_tensor[0][0]',
  (TFOpLambda)                                                            'tf.cast[0][0]']

 tf.math.reduce_sum (TFOpLa  (None,)                         0           ['tf.math.subtract_1[0][0]']
 mbda)

 tf.math.reduce_mean (TFOpL  ()                              0           ['tf.math.squared_difference[0
 ambda)                                                                  ][0]']

 tf.math.multiply (TFOpLamb  (None,)                         0           ['tf.math.reduce_sum[0][0]']
 da)
```

```
## Entrenar el modelo
vae.fit(embedded_sequence_input, embedded_sequence_output, epochs=epochs, batch_size=batch_size)
```

```
Epoch 1/30
65/65 [==============================] - 23s 321ms/step - loss: -17.5331
Epoch 2/30
65/65 [==============================] - 19s 296ms/step - loss: -19.9956
Epoch 3/30
65/65 [==============================] - 20s 314ms/step - loss: -19.9973
Epoch 4/30
65/65 [==============================] - 19s 296ms/step - loss: -19.9971
Epoch 5/30
```

```
65/65 [==============================] - 19s 293ms/step - loss: -19.9959
Epoch 6/30
65/65 [==============================] - 20s 315ms/step - loss: -19.9959
Epoch 7/30
65/65 [==============================] - 18s 279ms/step - loss: -19.9962
Epoch 8/30
65/65 [==============================] - 18s 278ms/step - loss: -19.9960
Epoch 9/30
65/65 [==============================] - 20s 306ms/step - loss: -19.9950
Epoch 10/30
65/65 [==============================] - 19s 293ms/step - loss: -19.9969
Epoch 11/30
65/65 [==============================] - 18s 274ms/step - loss: -19.9960
Epoch 12/30
65/65 [==============================] - 19s 296ms/step - loss: -19.9947
Epoch 13/30
65/65 [==============================] - 19s 293ms/step - loss: -19.9977
Epoch 14/30
65/65 [==============================] - 19s 300ms/step - loss: -19.9960
Epoch 15/30
65/65 [==============================] - 20s 305ms/step - loss: -19.9922
Epoch 16/30
65/65 [==============================] - 19s 293ms/step - loss: -19.9984
Epoch 17/30
65/65 [==============================] - 21s 318ms/step - loss: -19.9970
Epoch 18/30
65/65 [==============================] - 19s 293ms/step - loss: -19.9971
Epoch 19/30
65/65 [==============================] - 20s 307ms/step - loss: -19.9961
Epoch 20/30
65/65 [==============================] - 20s 310ms/step - loss: -19.9944
Epoch 21/30
65/65 [==============================] - 19s 294ms/step - loss: -19.9975
Epoch 22/30
65/65 [==============================] - 21s 320ms/step - loss: -19.9977
Epoch 23/30
65/65 [==============================] - 19s 289ms/step - loss: -19.9983
Epoch 24/30
65/65 [==============================] - 20s 301ms/step - loss: -19.9918
Epoch 25/30
65/65 [==============================] - 20s 311ms/step - loss: -19.9990
Epoch 26/30
```

```
Epoch 26/30
65/65 [==============================] - 19s 295ms/step - loss: -19.9982
Epoch 27/30
65/65 [==============================] - 25s 392ms/step - loss: -19.9963
Epoch 28/30
65/65 [==============================] - 18s 284ms/step - loss: -19.9992
Epoch 29/30
65/65 [==============================] - 20s 306ms/step - loss: -19.9973
```

```
# Obtener z_mean, z_log_var
z_mean_batch, z_log_var_batch = encoder.predict(embedded_sequence_input)

# Utilizar la función de muestreo para generar muestras de la distribución latente
latent_samples = sampling([z_mean_batch, z_log_var_batch])

# Decodificar las muestras generadas para obtener nuevos embeddings
decoded_sentence = decoder.predict(latent_samples)
```

```
260/260 [==============================] - 2s 7ms/step
260/260 [==============================] - 4s 15ms/step
```

Cálamos el modelito

```
# Crear el vocabulario inverso
inv_vocabulario = {idx: palabra for palabra, idx in vocabulario.items()}

decoded_sequence = []
for vector in decoded_sentence[1]:
    # Encuentra el índice del token más cercano en el espacio de embedding
    idx = np.argmax(vector)
    # Convierte el índice a token usando el diccionario inverso
    token = inv_vocabulario[idx]
    decoded_sequence.append(token)

reconstructed_text = ' '.join(decoded_sequence)
print("Texto reconstruido:", reconstructed_text)
```

```
print("Texto reconstruido:", reconstructed_text)
```

```
Texto reconstruido: cinemas cinemas cinemas cinemas cinemas cinemas cinemas cinemas stench planned planned
```

Malísimo pero lo intentó.

Tengo la teoría que el error que me da, es que estoy tratando los outputs del modelo como si fueran de los embeddings que usé pero quizá los vectores que me da no son compatibles, quizá tendría que hacer uno personalizado o no sé.

```
decoded_sentence[0]
```

```
array([[ 0.00342331, -0.0030201 ,  0.00167734, ..., -0.00552389,
        -0.00211306,  0.00721318],
       [ 0.00342331, -0.0030201 ,  0.00167734, ..., -0.00552389,
        -0.00211306,  0.00721318],
       [ 0.00342331, -0.0030201 ,  0.00167734, ..., -0.00552389,
        -0.00211306,  0.00721318],
       ...,
       [-0.01326514,  0.016698  ,  0.01247935, ..., -0.04293806,
         0.019782  ,  0.00447479],
       [-0.01325999,  0.0167598 ,  0.01250423, ..., -0.04318355,
         0.01987534,  0.00441948],
       [-0.01339806,  0.0168558 ,  0.01251846, ..., -0.04319974,
         0.01993513,  0.0044329 ]], dtype=float32)
```