Transformer Challenge

1. Get the the notebook from https://keras.io/examples/nlp/neural_machine_translation_with_transformer/
2. Make it run. Note that it uses the spa-eng file from the Anki site
3. Include code to save the model on disk so that you can use the pre-trained model
4. Include code to use the pre-trained embeddings from Stanford. Link at https://nlp.stanford.edu/projects/glove/
5. Include code to show the layer activations as the ones shown the notebook that shown during the lecture. Code at https://github.com/tensorflow/text/blob/master/docs/tutorials/transformer.ipynb
6. Work with the model to improve its performance. Things to try are:

- Use more than 30 epochs
- Change the number of ngrams
- Change the learning rate
- Change the optimizer
- Change the metric
- Explore how to use the BLUE (Bilingual Evaluation Understudy)
- Explore how to use the Rouge score

OPTIONAL: Get and run the code from https://keras.io/examples/nlp/neural_machine_translation_with_keras_hub/ to use the Rouge metric

Write a short report (5 pages) describing your work, results, comments Deadline: 03/22/2025 @ Noon, CDMX Time, using the Github page : https://github.com/camachojua/diplomado-ia/tree/main/python/src/student_submissions/Transformer

```
import os

os.environ["KERAS_BACKEND"] = "tensorflow"

import pathlib
import random
import string
import re
import numpy as np

import tensorflow.data as tf_data
import tensorflow.strings as tf_strings

import keras
from keras import layers
from keras import ops
from keras.layers import TextVectorization


text_file = keras.utils.get_file(
    fname="spa-eng.zip",
    origin="http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.
    extract=True,
)
text_file = pathlib.Path(text_file).parent / "spa-eng_extracted" / "spa-eng" /
```

```
Downloading data from http://storage.googleapis.com/download.tensorflow.org
2638744/2638744 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
```

Analizando los datos

Cada línea contiene una oración en inglés y su correspondiente oración en español. La oración en inglés es la secuencia de origen y la oración en español es la secuencia objetivo. Anteponemos el token "[start]" y añadimos el token "[end]" a la oración en español.

with open(text_file) as f:

```
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    eng, spa = line.split("\t")
    spa = "[start] " + spa + " [end]"
    text_pairs.append((eng, spa))
```

```
embed_dim = 256
latent_dim = 2048
num_heads = 8

encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="encoder_inputs
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)
encoder_outputs = TransformerEncoder(embed_dim, latent_dim, num_heads)(x)
encoder = keras.Model(encoder_inputs, encoder_outputs)

decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="decoder_inputs
encoded_seq_inputs = keras.Input(shape=(None, embed_dim), name="decoder_state_i
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, latent_dim, num_heads)([x, encoder_outputs])
x = layers.Dropout(0.5)(x)
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
decoder = keras.Model([decoder_inputs, encoded_seq_inputs], decoder_outputs)

transformer = keras.Model(
    {"encoder_inputs": encoder_inputs, "decoder_inputs": decoder_inputs},
    decoder_outputs,
    name="transformer",
)
```

Entrenando el modelo

```
from keras.optimizers import Adam

epochs = 6  # This should be at least 30 for convergence

transformer.summary()
custom_optimizer = Adam(learning_rate=0.001)####<------Tasa de aprendizaje para
transformer.compile(
    custom_optimizer,
    loss=keras.losses.SparseCategoricalCrossentropy(ignore_class=0),
    metrics=["accuracy"],
)
transformer.fit(train_ds, epochs=epochs, validation_data=val_ds)
```

Model: "transformer"

Layer (type)	Output Shape	Param #	Con
encoder_inputs (InputLayer)	(None, None)	0	–
decoder_inputs (InputLayer)	(None, None)	0	–
positional_embedding (PositionalEmbedding)	(None, None, 256)	3,845,120	enc
not_equal (NotEqual)	(None, None)	0	enc
positional_embedding_1 (PositionalEmbedding)	(None, None, 256)	3,845,120	dec
transformer_encoder (TransformerEncoder)	(None, None, 256)	3,155,456	pos not
not_equal_1 (NotEqual)	(None, None)	0	dec
transformer_decoder (TransformerDecoder)	(None, None, 256)	5,259,520	pos tra not not
dropout_3 (Dropout)	(None, None, 256)	0	tra
dense_4 (Dense)	(None, None, 15000)	3,855,000	dro

Total params: 19,960,216 (76.14 MB)
Trainable params: 19,960,216 (76.14 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 108s 69ms/step - accuracy: 0.0776 - loss: 5.
Epoch 2/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 124s 62ms/step - accuracy: 0.1384 - loss: 3.
Epoch 3/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 79s 61ms/step - accuracy: 0.1775 - loss: 2.8
Epoch 4/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 83s 62ms/step - accuracy: 0.2031 - loss: 2.3
Epoch 5/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 82s 62ms/step - accuracy: 0.2200 - loss: 1.9
Epoch 6/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 81s 63ms/step - accuracy: 0.2327 - loss: 1.7
<keras.src.callbacks.history.History at 0x7f8b823e9ad0>
```

4. Modificaciones a la clase `PositionalEmbedding` para cargar los embeddings de GloVe, se crea la clase `PositionalEmbeddingGlove`

Descripción

Se realizaron modificaciones a la clase `PositionalEmbedding` para que no solo maneje las representaciones posicionales, sino también para cargar embeddings preentrenados de GloVe. Esto mejora la calidad de las representaciones de las palabras al aprovechar los vectores de palabras preentrenados de un modelo de embeddings de palabras como GloVe.

Modificaciones

1. **Carga de los embeddings de GloVe**:
    - Los embeddings de GloVe fueron cargados desde un archivo de texto que contiene los vectores preentrenados.
    - Estos vectores fueron almacenados en un diccionario, donde las claves son las palabras y los valores son sus representaciones vectoriales.
2. **Integración de los embeddings de GloVe en la capa de embedding**:
    - En lugar de usar un embedding aleatorio, se cargó una matriz de embeddings preentrenados y se asignó a la capa de embeddings de la clase `PositionalEmbedding`.
    - Solo las palabras que estén en el vocabulario de entrada serán representadas con los embeddings de GloVe. Las palabras fuera del vocabulario son representadas con un vector de ceros o con un valor de inicialización predeterminado.

```
class PositionalEmbeddingGlove(layers.Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim, embedding_matrix
        super().__init__(**kwargs)

        self.token_embeddings = layers.Embedding(
            input_dim=vocab_size,
            output_dim=embed_dim,
            weights=[embedding_matrix] if embedding_matrix is not None else Non
            trainable=False if embedding_matrix is not None else True
        )

        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=embed_dim
        )

        self.sequence_length = sequence_length
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim

    def call(self, inputs):
        length = ops.shape(inputs)[-1]
        positions = ops.arange(0, length, 1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        return ops.not_equal(inputs, 0)

    def get_config(self):
        config = super().get_config()
        config.update(
            {
                "sequence_length": self.sequence_length,
                "vocab_size": self.vocab_size,
                "embed_dim": self.embed_dim,
            }
        )
        return config
```

Cargar y usar los embeddings preentrenados de GloVe

Descripción

En esta celda se cargan los embeddings preentrenados de GloVe y se integran en la red neuronal. Los embeddings de GloVe son utilizados para representar las palabras en el vocabulario antes de pasar por el modelo Transformer.

Pasos realizados

1. **Carga de los embeddings de GloVe**:
    - Se carga el archivo de GloVe (`glove.6B.50d.txt`) que contiene vectores preentrenados de palabras en un diccionario llamado `embeddings_index`. Cada palabra se mapea a su correspondiente vector de características.
2. **Creación de la matriz de embeddings**:
    - Se construye una matriz llamada `embedding_matrix`, la cual tiene un tamaño igual al del vocabulario de entrada. Cada palabra en el vocabulario se mapea al vector de GloVe correspondiente si está presente en el diccionario `embeddings_index`.
3. **Modificación del modelo**:
    - Se crea una capa personalizada llamada `PositionalEmbeddingGlove`, la cual recibe la matriz de embeddings de GloVe para ser utilizada en el modelo.
    - El encoder y decoder del modelo Transformer ahora utilizan esta capa de embeddings para representar las palabras de manera más informada.

```
# Ruta al archivo GloVe
glove_path = "glove.6B.50d.txt"

# Cargar GloVe en un diccionario
embeddings_index = {}
with open(glove_path, encoding="utf-8") as f:
    for line in f:
        values = line.split()
        word = values[0]  # Primera palabra en la línea
        coefs = np.asarray(values[1:], dtype="float32")  # Resto son los valore
        embeddings_index[word] = coefs

print(f"Se cargaron {len(embeddings_index)} palabras preentrenadas.")

spa_vocab = spa_vectorization.get_vocabulary()
```

```
epochs = 6

transformer2.summary()
transformer2.compile(
    "rmsprop",
    loss=keras.losses.SparseCategoricalCrossentropy(ignore_class=0),
    metrics=["accuracy"],
)
transformer2.fit(train_ds, epochs=epochs, validation_data=val_ds)
```

Model: "transformer"

Layer (type)	Output Shape	Param #	Con
decoder_inputs (InputLayer)	(None, None)	0	–
encoder_inputs (InputLayer)	(None, None)	0	–
positional_embedding_glo… (PositionalEmbeddingGlov…	(None, None, 50)	751,000	enc dec
not_equal_2 (NotEqual)	(None, None)	0	enc
transformer_encoder_1 (TransformerEncoder)	(None, None, 50)	288,348	pos not
not_equal_3 (NotEqual)	(None, None)	0	dec
transformer_decoder_1 (TransformerDecoder)	(None, None, 50)	369,698	pos tra not not
dropout_7 (Dropout)	(None, None, 50)	0	tra
dense_9 (Dense)	(None, None, 15000)	765,000	dro

Total params: 2,174,046 (8.29 MB)
Trainable params: 1,424,046 (5.43 MB)
Non-trainable params: 750,000 (2.86 MB)

```
Epoch 1/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 52s 29ms/step - accuracy: 0.0596 - loss: 6.4
Epoch 2/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 24s 19ms/step - accuracy: 0.1015 - loss: 4.9
Epoch 3/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 24s 18ms/step - accuracy: 0.1198 - loss: 4.5
Epoch 4/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 24s 19ms/step - accuracy: 0.1310 - loss: 4.3
Epoch 5/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 24s 19ms/step - accuracy: 0.1378 - loss: 4.2
Epoch 6/6
1302/1302 ━━━━━━━━━━━━━━━━━━━━ 24s 18ms/step - accuracy: 0.1419 - loss: 4.1
<keras.src.callbacks.history.History at 0x7f8b24102e10>
```

```
%%capture
!pip install nltk rouge-score
```

Métricas BLEU y ROUGE

Las métricas BLEU y ROUGE son utilizadas para evaluar la calidad de los sistemas de traducción automática, comparando las traducciones generadas por el modelo con las traducciones de referencia.

BLEU (Bilingual Evaluation Understudy)

- **BLEU** mide la precisión n-grama de una traducción, comparando la cantidad de n-gramas coincidentes entre la traducción generada y las referencias.
- Se utiliza un **suavizado** para evitar ceros en los cálculos de n-gramas raros o ausentes.
- El cálculo de BLEU se realiza usando la función `corpus_bleu` de la librería `nltk`.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

- **ROUGE** evalúa la calidad de la traducción basándose en la comparación de n-gramas y subsecuencias con las referencias.
- Se considera principalmente el **recall** y la **longitud L** (ROUGE-L), que mide la longitud máxima de la subsecuencia común más larga.
- Se calcula usando la librería `rouge_score`.

```
#Definir la función de decodificación
spa_vocab = spa_vectorization.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = eng_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = spa_vectorization([decoded_sentence])[:, :-1
        predictions = transformer({
            "encoder_inputs": tokenized_input_sentence,
            "decoder_inputs": tokenized_target_sentence
        })
        sampled_token_index = ops.convert_to_numpy(ops.argmax(predictions[0, i,
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

```
print(f"ROUGE-1: {rouge1:.4f}, ROUGE-L: {rougeL:.4f}")
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.

Evaluación de métricas de traducción:
BLEU score: 0.1341316843687854
ROUGE-1: 0.5559, ROUGE-L: 0.5478
```

Resultados 6.2 n-grams

Solo modificamos *eng_vectorization* y agregamos **ngrams=m** para algun $m \in \mathbb{N}$.

Número de n-grams	Accuracy	Loss	Val Accuracy	Val Loss
2	0.2392	2.0617	0.2311	2.2354
4	0.2361	2.1264	0.2285	2.2466
6	0.2354	2.1343	0.2301	2.2569

Por lo tanto, tomamos $n - grams = 2$ ya que es el que nos dio un mayor valor de validación en Accuracy.

Resultados 6.3 tasa de aprendizaje

Continuamos con la tasa de aprendizaje lr, agregamos el optimizador ADAM para modificar su tasa de aprendizaje:

ADAM(lr)	Accuracy	Loss	Val Accuracy	Val Loss
lr=0.001	0.2289	1.7962	0.2208	2.0142
lr=0.005	0.2051	2.1636	0.2031	2.2623
lr=0.01	0.0502	5.9670	0.0499	5.8368

Podemos notar como a medida de que aumentamos la tasa de aprendizaje del optimizador ADAM, la precisión disminuye y tambien aumenta la perdida del modelo. Esto último ocurre tanto en el entrenamiento como en la validación. Por lo tanto, se deben de balancear estas dos métricas como es el caso de la tasa de aprendizaje de $lr = 0.001$ o $lr = 0.005$ donde valores cercanos a esos valores darían mejores resultados tanto en la Accuracy como en la perdida, dependiendo que se quiera mejorar más.

6.4 Optimizador

Cambiamos el optimizador cargando algun otro de keras y rn *compile* agregamos estos optimizadores los cuale son los siguientes:

Optimizador(lr)	Accuracy	Loss	Val Accuracy	Val Loss
RMSprop(0.001)	0.0503	5.8545	0.0504	5.7475
AdamW(0.001)	0.2481	1.3703	0.2293	1.8646
Lion	0.0590	5.6973	0.0599	5.5801

Nuevamente, la variacion de ADAM, ADAMW fue el que obtuvo mejores resultados con errores de entrenamiento más bajos que el ADAM normal.

6.5 Métrica

Al cambiar la métrica a *sparse categorical accuracy*, esta baja utilizando el ADAM(lr=0.001) de la siguiente manera:

Loss	Sparse Categorical Accuracy	Val Loss	Val Sparse Categorical Accuracy
5.6287	0.0605	5.5408	0.0601

Por lo cual no es conveniente cambiarla ya que aumenta la perdida.

6.6 y 6.7 Metricas BLUE (Bilingual Evaluation Understudy) y Rouge

Métricas de Evaluación para Traducción Automática/Texto Generado

BLEU (Bilingual Evaluation Understudy)

- Mide la similitud entre texto generado y referencias humanas usando *coincidencia de n-gramas* (usualmente 1-4 gramas).
- **Usos típicos**:
    - Evaluación de traducción automática
    - Sistemas de conversión texto-a-texto
- **Limitaciones**:
    - No considera significado semántico
    - Pobre desempeño con idiomas flexibles
    - Favorece traducciones literales

ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

-Mide la adecuación del contenido mediante *recuperación de unidades léxicas*.

- **Usos típicos**:
    - Evaluación de resúmenes
    - Sistemas de generación de texto
- **Limitaciones**:
    - No evalúa coherencia
    - Sensible a redundancias

    - Ignora relaciones semánticas

Característica	BLEU	ROUGE
Enfoque principal	Precisión	Recall
Ideal para	Traducción	Resumen/Generación
Sensibilidad a longitud	Penaliza cortos	Menos sensible
Unidad de análisis	N-gramas exactos	Subsecuencias
Tiempo de cálculo	Rápido	Moderado
Métrica oficial en	WMT (traducción)	DUC (resumen)

Pra interpretar los scores:

- **BLEU**:
    - mayor a 0.4: Excelente
    - 0.3-0.4: Bueno
    - menor a 0.2: Pobre
- **ROUGE**:
    - mayor a 0.5: Alta calidad
    - 0.3-0.5: Aceptable
    - menor a 0.2: Baja calidad

Para nuestro ejemplo de 6 epocas con un optimizador Adam con *lr=0.001* y *ngrams=2* obtuvimos:

- BLEU score: 0.1341316843687854.
- ROUGE-L: 0.5478.

Donde los modelos son pobres debido a las pocas epocas que se usaron.

```
#Guardar el modelo
transformer.save("modelo_transformer.keras")

#Guardar los pesos
transformer.save("transformer_pesos.h5")

# Guardar modelo con pretrained embedding
transformer2.save("transformer_model_glove.keras")

transformer = keras.models.load_model(
    "transformer_model_rnd.keras",
    custom_objects={
        "encoder_inputs": encoder_inputs,
        "decoder_inputs": decoder_inputs,
        "TransformerEncoder": TransformerEncoder,
        "TransformerDecoder": TransformerDecoder,
        "PositionalEmbedding": PositionalEmbedding,
    },
)

transformer_glove = keras.models.load_model(
    "transformer_model_glove.keras",
    custom_objects={
        "encoder_inputs": encoder_inputs,
        "decoder_inputs": decoder_inputs,
        "TransformerEncoder": TransformerEncoder,
        "TransformerDecoder": TransformerDecoder,
        "PositionalEmbeddingGlove": PositionalEmbeddingGlove,
    },
)
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` o
```

Decoding test sentences