UNIVERSIDAD SAN CARLOS DE GUATEMALA

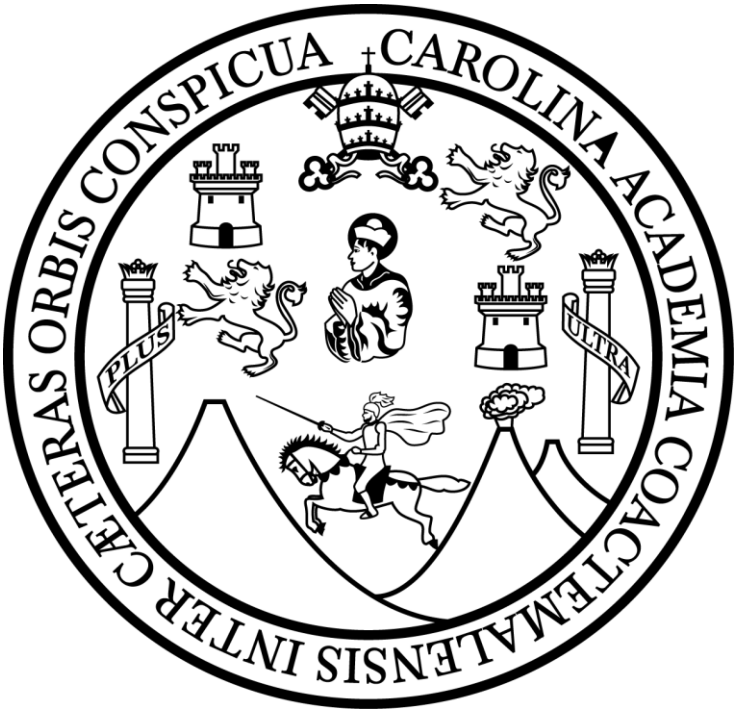
FACULTAD DE INGENIERIA

ESCUELA DE CIENCIAS Y SISTEMAS

INTELIGENCIA ARTIFICIAL 1

SECCIÓN A

ESCUELA DE VACACIONES, DICIEMBRE 2024



PROYECTO ÚNICO – FASE #1

GRUPO #5

POR:

3134216330901	MORALES XICARA, ERICK DANIEL
3146392170901	HERNÁNEZ SAPÓN, LEVÍ ISAAC
3348212820901	SÁNCHEZ SANTOS, LUIS FERNANDO

GUATEMALA, QUETZALTENANGO, 10/12/2024

INDICE

MANUAL TÉCNICO

Descripción del Proyecto

El primer proyecto asignado para el curso de Inteligencia Artificial 1, tiene como objetivo principal la creación de un modelo de inteligencia artificial, el cual sea capaz de poder procesar entradas de textos en idioma español y este pueda responder a las preguntas planteadas por el usuario de forma coherente, funcional y especifica. Para la solución de dicho problema, se optará por utiliza tecnologías que han sido implementadas y basadas en el lenguaje de programación Python y este será entregado y desarrollado mediante un modelo incremental, dado que se le implementaran mejoras continuas al modelo de inteligencia artificial, lo cual garantiza que el resultado pueda llegar a cumplir de manera satisfactoria con los requisitos que ha impuesto el cliente en cada fase correspondiente al desarrollo del sistema.

Alcance del Proyecto

Dado que el modelo a desarrollar será implementado de manera incremental a lo largo del mes, se optará por emplear una metodología que garantice y priorice la evaluación y mejora continua. Para ello se han identificado los siguientes puntos clave:

- Investigación y Selección de Tecnologías

  Durante la primera semana de desarrollo, se llevará a cabo una investigación para poder identificar las bibliotecas o frameworks compatibles con el lenguaje de programación Python, que nos permita poder crear y desarrollar un modelo de IA eficiente y congruente con las especificaciones brindadas por el cliente. La selección se cimentará en criterios como: facilidad de uso, soporte técnico de las librerías en caso de presentar inconvenientes, documentación amplia y capacidad de poder gestionar entrenamientos efectivos.

- Desarrollo del modelo de inteligencia artificial

  El modelo será diseñado para que este pueda ser capaz de poder recibir entradas textuales, analizar el contenido semántico y poder generar respuestas contextualizadas. Este proceso implicará:

  - Se realizará un entrenamiento inicial empleando Datasets en español, que le permitan a la IA poder interpretar interacciones comunes o relevantes para contextos cotidianos que pudiesen llegar a solicitar los usuarios.
  - Configuración de un pipeline de procesamiento de textos para tareas como tokenización, análisis de contexto a las peticiones del usuario y generación congruente en base al contexto para las respuestas brindadas por la IA.

- Implementación de Interfaz de Usuario

  Para que pueda existir una interacción entre el usuario y el modelo de IA desarrollado, se desarrollara una interfaz gráfica web, utilizando el framework de Angular, cuya implementación e interacción que tendrá el usuario a través de la UI, será detallada más adelante.

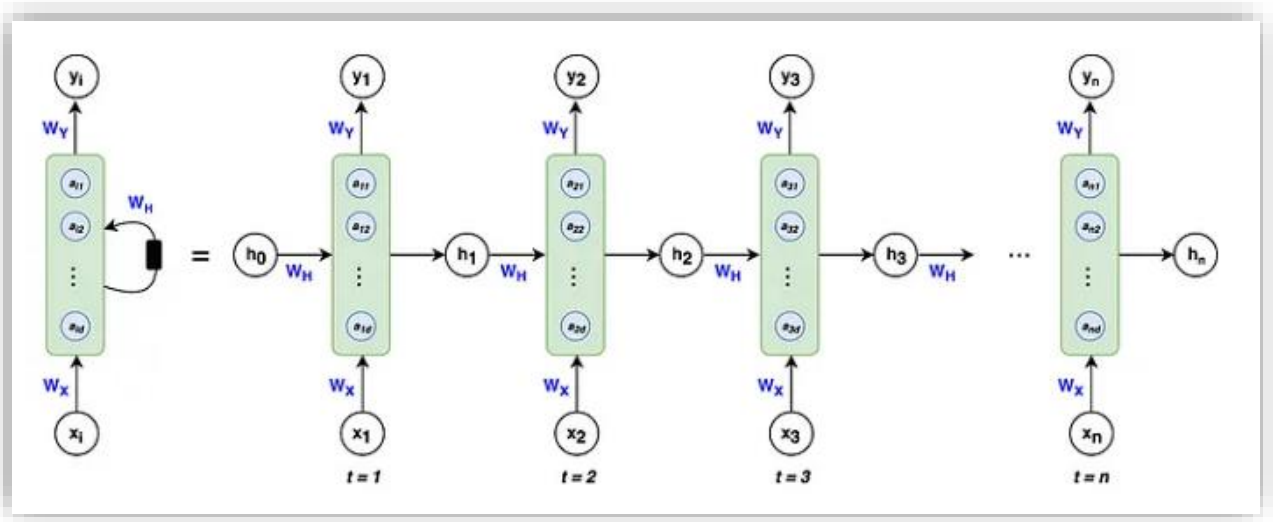
Enfoque para el desarrollo de la IA

Para poder desarrollar el modelo de inteligencia artificia, se optará por implementar una arquitectura Sequence-to-Sequence, la cual está basada en redes neuronales recurrentes (RNN) con capas Long Short-Term Memory (LSTM). Dicho enfoque que será implementado es ideal para poder realizar procesamiento del lenguaje natural (NLP), en el cual se presenta una relación entre la secuencia de entrada (pregunta realizada por el usuario) y una secuencia de salida (respuesta brindada por el modelo de IA).

Redes Neuronales Recurrentes (RNN)

Las redes neuronales recurrentes son un tipo especializado de red neuronal diseñado para poder ser utilizada en datos secuenciales. Su principal característica es la capacidad de poder recordar información previa en la entrada y utilizarla en las decisiones actuales, lo cual lo hace ideal para tareas donde el orden que requieren los datos es importante, así como el análisis del lenguaje natural, series temporales y reconocimiento de patrones en audio o video. Las redes neuronales funcionan de la siguiente manera:

- Estado Oculto: Este estado funciona como una memoria que almacena información sobre las entradas previas en la secuencia, lo cual le permite a la red poder tomar decisiones informadas en datos pasados
- Pesos Compartidos: Las RNN emplean pesos para cada elemento de la secuencia, lo que ayuda a reducir la complejidad del modelo y facilita la generación a diferentes tamaños de secuencias
- Procesamiento secuencial: Cada salida depende no solo del elemento actual que esta siendo analizado, sino también del estado oculto, el cual refleja la información acumulada de entradas anteriores
- Estructura en bucle: Las RNN tienen conexiones recurrentes que permiten poder pasar información de un estado al siguiente dentro de la misma red.



Long Short-Term Memory (LSTM)

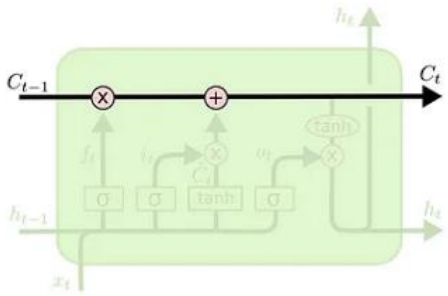
La aplicación de redes neuronales recurrentes en un modelo de inteligencia artificial, presentan un problema clave, el cual es, la incapacidad de la red para poder manejar secuencias largas debido al problema del gradiente desaparecido. Este inconveniente se presente cuando en las actualizaciones del peso durante el entrenamiento de la IA disminuyen drásticamente, lo cual provoca que el error se propague a través de las diferentes capas de la red neuronal. Y nos conlleva a las siguientes problemáticas:

- Las RNN no pueden aprender relaciones a largo plazo en secuencias
- Los datos más antiguos en la secuencia terminan perdiendo relevancia para su desarrollo.

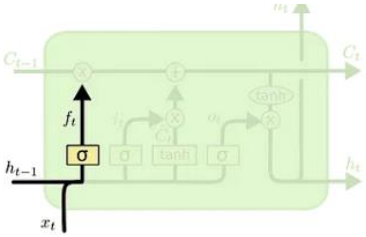
Para poder llevar a cabo una solución eficiente a dicha problemática, se opta por implementar las Long Short-Term Memory (LSTM), las cuales son una arquitectura avanzada de redes neuronales diseñadas para abordar esta problemática. Para ello se opta por introducir u mecanismo para poder olvidar la información irrelevante y recordar datos esenciales; lo cual conlleva a que las LSTMs puedan tomar decisiones basadas en contexto de la información previa y actual.

Las LSTMs extienden las RNN tradicionales mediante componente adicionales que las hacen más potentes a través de múltiples componentes (estado de celda y puertas) para poder controlar como la información se retiene, se olvida o se utiliza:

- Estado de celda: Esta no es más que una cinta transportadora que atraviesa toda la unidad LSTM y transporta la información relevante a lo largo de la secuencia, lo cual permite poder recordar información relevante, olvidar datos irrelevantes y añadir nueva información. Para llevar a cabo dichos procesos utilizan las siguientes operaciones:
  - Multiplicación punto a punto (X): Determina que información olvidar (0) o retener (1)
  - Suma (+); Agrega información nueva seleccionada:

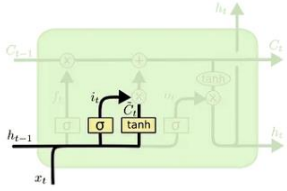


- Puerta de Olvido: Está decide qué información del estado previo debe olvidarse, para llevar a cabo dicha decisión, debe realizar el siguiente mecanismo.
  - Se debe de calcular el producto punto entre la salida anterior y la entrada actual
  - Se aplica una función *sigmodid*, la cual produce valores entre 0 y 1
  - El valor cercano a 0 implica olvidar complementa la información ingresada; mientras que un valor cercano a 1, nos incita a conservar dicha información.



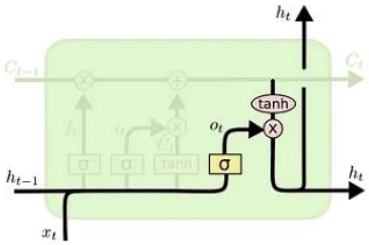
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- Puerta de entrada: Esta añada información al estado de la celda, para llevar cabo la adición de información, se debe llevar a cabo el siguiente mecanismo:
  - Una capa *sigmodid* decide que valores serán actualizados.
  - Una función *tanh* genera valores candidatos que podrían llegar a ser añadidos al estado.
  - Finalmente, se realiza una combinación entre las salidas con la finalidad de poder actualizar el estado actual que posee la celda.



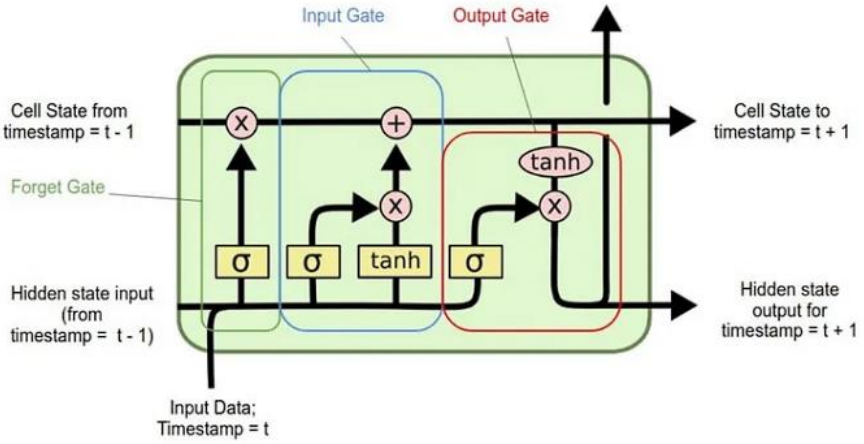
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

- Puerta de salida: Esta decide que parte del estado de la celda se usara como salida, para tomar dicha decisión, se debe llevar a cabo el siguiente mecanismo:
  - Una capa *sigmodid* filtra la información relevante
  - Una capa *tanh* ajusta los valores del estado de la celda entre -1 y 1
  - La salida final se calcula multiplicando ambas capas.



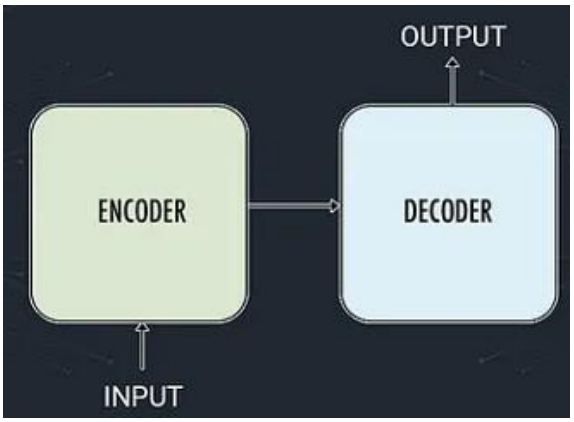
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Este funcionamiento en conjunto permite procesar la información secuencialmente a través de estas puertas, lo cual le permite poder olvidar datos irrelevantes, actualizar la memoria y generar una salida a través de la selección de que partes del estado de celda serán útiles para la predicción actual.



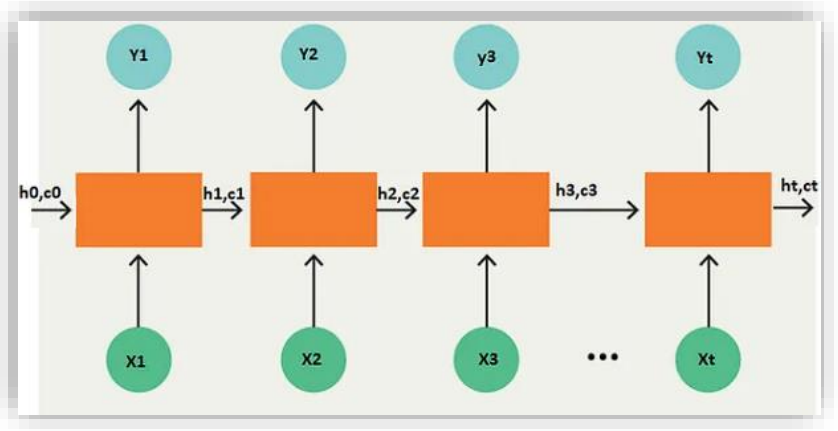
Sequence-to-Sequence Model

Los modelos Sequence-to-Sequence son redes neuronales que son diseñadas para poder transformar una secuencia de entrada en una secuencia de salida, lo cual la hace ideal para poder asignarle tareas como traducción automática, subtitulado de videos, generación de texto a partir de imágenes y sistemas de preguntas y respuestas.

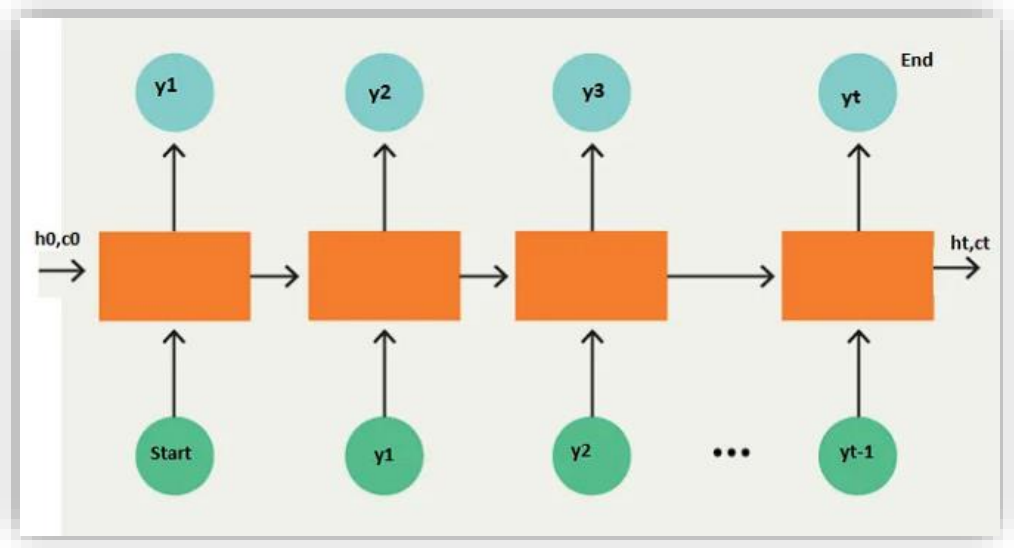


El enfoque estándar para poder construir los modelos Seq2Seq es la arquitectura Encoder-Decoder. La cual permite poder dividir la arquitectura en dos partes elementales.

- Encoder (Codificador): Este se encarga de poder procesar la secuencia de entrada y poder generar un vector de contexto que permite poder generar un vector de contexto, el cual se encarga de poder resumir la información, para llevar a cabo dicho procedimiento se debe llevar a cabo el siguiente proceso:
  - Se lee una secuencia de entrada X={x1,x2,...,xt} paso a paso.
  - Se produce un vector de contexto (estados internos) compuesto por un estado oculto y un estado de salida
  - Se procede a Descartar salidas directas del encoder y únicamente se conserva sus estados internos finales.



- Decoder (Decodificador): Este se encarga de poder generar una secuencia de salida a partir del contexto proporcionado por el encoder. Para llevar a cabo dicho procedimiento se debe llevar a cabo el siguiente proceso:
  - El estado inicial del decodificador (h_0, c_0) se establece con los estados finales del encoder.
  - A partir un toque especial de inicio, denominado Start, genera la secuencia Y = {y_1, y_2, …, y_t).
  - En cada paso el decodificador utiliza su estado interno actual y el token previo para predecir el siguiente token.



Para que el modelo pueda ser entrenado se emplean los tokens de salida brindados por el decorder y en cada paso se procede a realizar una comparación con los tokens reales, para poder llevar a cabo el cálculo de una perdida. Lo cual conlleva a que los errores puedan llegar a propagarse para poder ajustar los pesos a través de toda la red neuronal. Para llevar a cabo dicho procedimiento, se lleva a cabo un flojo general en base al tiempo de inferencia que requiere el entrenamiento del modelo:

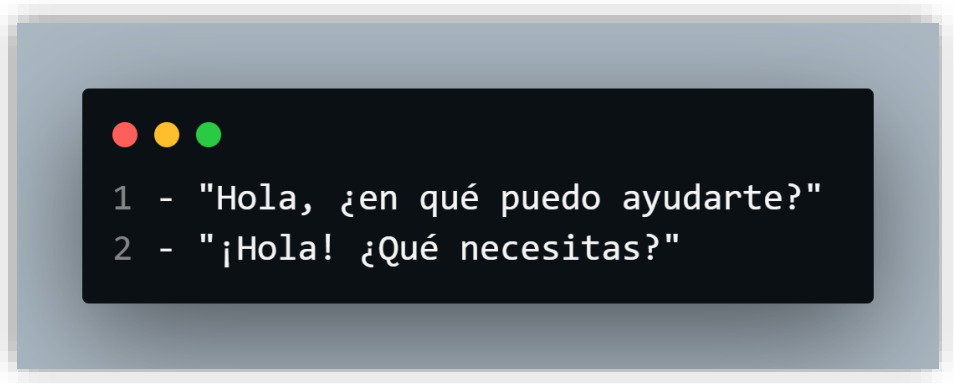
1. El token *start* se usa como una entrada inicial para el decodificador
2. El decodificador genera un token a la vez
3. Cada token predicho se utiliza como entrada al siguiente paso
4. El proceso termina cuando el decodificado genera el token *end*

Intents

Los Intents son aplicados en el ámbito del procesamiento del lenguaje natural, es una representación estructurada de la intención que puede llegar a tener el usuario al comunicarse con un sistema. Se utiliza principalmente al aplicarlo con chatbots, asistentes virtuales y sistemas de clasificación del texto. Para la maquetación de un Intent son necesarios los siguientes componentes:

- Nombre del Intent: Una etiqueta o identificador único que describe la intención que pueda llegar a tener el usuario.
- Patrones de entrada: Estas son representaciones de frases que presenten ejemplos de cómo un usuario podría llegar a expresar una intensión, como podría ser en la siguiente entrada:

- Respuestas: Estas son salidas generadas por el sistema para poder identificar de forma adecuada un intent, como pudiese llegar a ser las pertenecientes a un saludo:



- Datos adicionales: Se le puede asignar a estos modelos información extra que el sistema pudiese llegar a necesitar para poder cumplir con la intención que pueda llegar a pretender el usuario que se le presente, como pueden ser parámetros o entidades.

Para poder cumplir con las especificaciones de la creación del modelo, es posible utilizar **Tenserflow**,el cual es utilizado para poder construir y entrenar modelos que puedan clasificar intents. El modelo pretende asociar patrones de texto (entrada) con un intent especifico (salida), para poder cumplir con dicho proceso general implica:

1. Procesamiento de Data: Se procede a convertir cada una de las frases de entrada en un entorno de formato que el modelo pueda entender (vectores numéricos)
2. Definición del modelo: Se procede a diseñar una red neuronal que clasifique las frases en intents.
3. Entrenamiento del modelo: Se ajusta el modelo usando datos de entrenamiento
4. Predicción: Se utiliza el modelo entrenado para poder clasificar nuevas frases de entrada.

Arquitectura para el desarrollo de la IA

La arquitectura se divide en dos componentes principales: el codificador y el decodificador, los cuales trabajan en conjunto con la finalidad de poder comprender el contexto de la entrada y poder generar una respuesta coherente basada en el contexto de la pregunta realizada por el usuario.

Adicionalmente fueron implementadas capas de embeddings para poder representar palabras como vectores densos en un espacio semántico y capas densas para la predicción de tokens en la secuencia de salida.

El codificador procesa la entrada (pregunta) y genera un conjunto de estados internos que representan el contexto de la secuencia. El proceso que realiza para poder llevar a cabo dicha secuencia es la siguiente:

- Input Layer: Recibe secuencias tokenizadas y las convierte en embeddings utilizando una capa de tipo Embedding.
- LSTM Layer: Genera representaciones contextuales y produce los estados finales de la celda (state_h, state_c) que son pasados al decodificador.
- Output: Los estados finales del codificador (state_h, state_c) que encapsulan la información semántica de la entrada.

El decodificador genera una respuesta basada en los estados recibidos del codificador y los datos de entrada del decodificador. El proceso que realiza para poder llevar a cabo dicha secuencia es la siguiente:

- Input Layer: Recibe las secuencias de inicio (<START>) como entrada inicial.
- Embedding Layer: Similar al codificador, convierte tokens en vectores densos.
- LSTM Layer: Genera una secuencia de estados y predicciones para cada token de la respuesta.
- Dense Layer: Aplica una función de softmax para producir una distribución de probabilidad sobre el vocabulario, prediciendo la siguiente palabra.

Implementación y Flujo

1. Procesamiento de datos
   - ❖ Las preguntas y respuestas fueron extraídas de archivos YAML y limpiadas mediante expresiones regulares para normalizar texto.
   - ❖ Se agregaron etiquetas especiales <START> y <END> a las respuestas, definiendo puntos claros para el inicio y fin de las secuencias de salida.

2. Tokenización:
    - ❖ Se utilizó un tokenizer para convertir palabras en índices enteros que representan la posición de cada palabra en el vocabulario.
    - ❖ La tokenización incluyó palabras de preguntas y respuestas para garantizar una representación uniforme del vocabulario.

3. Entrenamiento del Modelo:
    - ❖ Entrada del Codificador: Estas se brindan a través de las secuencias de preguntas preprocesadas.
    - ❖ Entrada del Decodificador: Estas son las respuestas que poseen la etiqueta <START>.
    - ❖ Salida Esperada: Estas respuestas son tokenizadas para poder ser empleadas por el modelo de IA
4. Inferencia
    - ❖ Se creó un modelo separado para el codificador, el cual toma una pregunta y devuelve los estados contextuales.
    - ❖ El decodificador se implementó como un modelo independiente para generar respuestas palabra por palabra, utilizando los estados iniciales del codificador.
    - ❖ El flujo de predicción utiliza una secuencia vacía que es actualizada iterativamente con las palabras generadas, hasta alcanzar <END> o el límite de longitud máxima.

Arquitectura Detallada

Codificador

El codificador transforma una secuencia de entrada en un conjunto de estados internos que encapsulan su significado semántico. La cual debe de seguir el siguiente flujo:

- o La entrada se tokeniza y se transforma en vectores mediante una capa de embeddings.
- o Los embeddings pasan a través de una capa LSTM que produce los estados ocultos (state_h) y los estados de la celda (state_c).
- o Estos estados resumen el contexto de la entrada y son utilizados por el decodificador.

```python
# Entrada del codificador
encoder_inputs = tf.keras.layers.Input(shape=(maxlen_questions,))  # Secuencia tokenizada
encoder_embedding = tf.keras.layers.Embedding(VOCAB_SIZE, 200, mask_zero=True)(encoder_inputs)

# Capa LSTM del codificador
encoder_outputs, state_h, state_c = tf.keras.layers.LSTM(200, return_state=True)(encoder_embedding)

# Estados finales del codificador
encoder_states = [state_h, state_c]
```

Decodificador

El decodificador genera la secuencia de salida palabra por palabra, utilizando los estados del codificador como contexto inicial. La cual debe de seguir el siguiente flujo:

- La entrada inicial es la etiqueta <START> tokenizada.
- Se pasa por una capa de embeddings para obtener vectores densos.
- Una capa LSTM genera predicciones basadas en los estados del codificador y las entradas previas del decodificador
- Finalmente, una capa densa con activación softmax produce una distribución de probabilidad sobre el vocabulario para predecir la siguiente palabra.

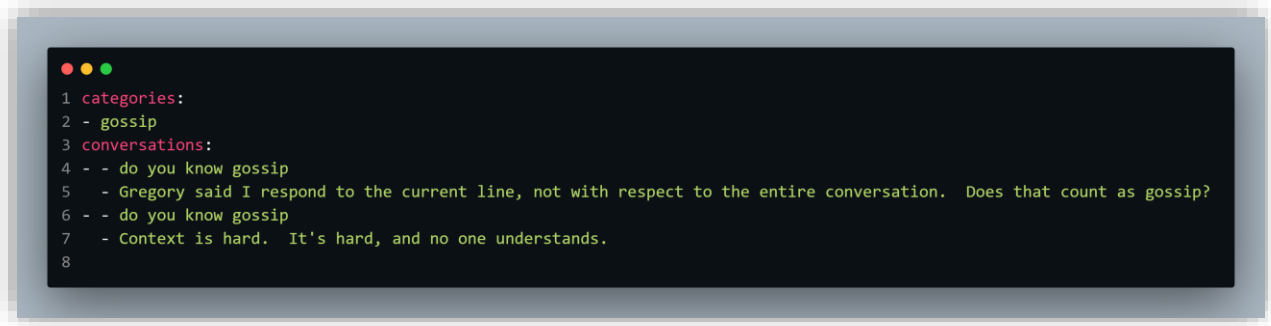
```python
# Entrada del decodificador
decoder_inputs = tf.keras.layers.Input(shape=(maxlen_answers,))  # Secuencia inicial con <START>
decoder_embedding = tf.keras.layers.Embedding(VOCAB_SIZE, 200, mask_zero=True)(decoder_inputs)

# LSTM del decodificador
decoder_lstm = tf.keras.layers.LSTM(200, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_embedding, initial_state=encoder_states)

# Capa densa para predecir el próximo token
decoder_dense = tf.keras.layers.Dense(VOCAB_SIZE, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

```

Fase de entrenamiento para la IA

Data Entrada

Para que se pueda realizar el entrenamiento de la IA, se opto por utilizar archivos YAML, los cuales tienen una estructura clara y se emplearon para almacenar datos de conversaciones con categorías temáticas específicas y preguntas/respuestas asociadas. Estos archivos contienen dos elementos principales:

- Categorías: Estas incluyen listas de temas bajo los cuales serán agrupadas las conversaciones para poder entrenar el modelo, para ello cada categoría representa un dominio semántico en específico.
- Conversaciones: Estas incluyen una lista de pares pregunta-respuestas, en la cual, cada conversación estará estructurada como una lista de sublistas.



```yaml
categories:
- gossip
conversations:
- - do you know gossip
  - Gregory said I respond to the current line, not with respect to the entire conversation.  Does that count as gossip?
- - do you know gossip
  - Context is hard.  It's hard, and no one understands.

```

Carga y Estructuración de datos

Para poder realizar una estructuración de los datos es necesario ubicar donde se encuentran los archivos YAML para que el modelo pueda ser entrenado. Para ello dentro del código se realizo la siguiente estructuración para realizar la carga de la Data.

- La variable *dir_path* se encargará de apuntar la dirección donde se encuentran los archivos YAML.
- A la variable *files_list* se le asignara el valor del método os.listdir, el cual se encarga de recuperar una lista de archivo en el directorio.
- Cada archivo se abre y se procede a cargarlo en memoria, usando el método *yaml.safe_load*.

Una vez que se haya procedido a asignarle una estructura la data, es necesario asignarles una estructura a las conversaciones, dado que las conversaciones se encuentran bajo la clave *conversations* de los archivos YAML. Dado que cada conversación tiene un formato de lista donde el primer elemento es una pregunta y los siguientes son respuestas, por lo cual es necesario realizar la siguiente categorización

- Si una conversación tiene varias preguntas, se concatenen en una sola cadena, utilizando el método *.join()*.
- Una vez hecha esa distinción, se almacenan preguntas y respuestas en listas paralelas (*questions* y *answers)*.

```
dir_path = './kaggle/input/'
files_list = os.listdir(dir_path + os.sep)

questions, answers = [], []

for filepath in files_list:
    with open(dir_path + os.sep + filepath, 'rb') as file_:
        docs = yaml.safe_load(file_)
        conversations = docs['conversations']
        for con in conversations:
            if len(con) > 2:
                questions.append(con[0])
                replies = con[1:]
                ans = ' '.join(replies)
                answers.append(ans)
            elif len(con) > 1:
                questions.append(con[0])
                answers.append(con[1])
```

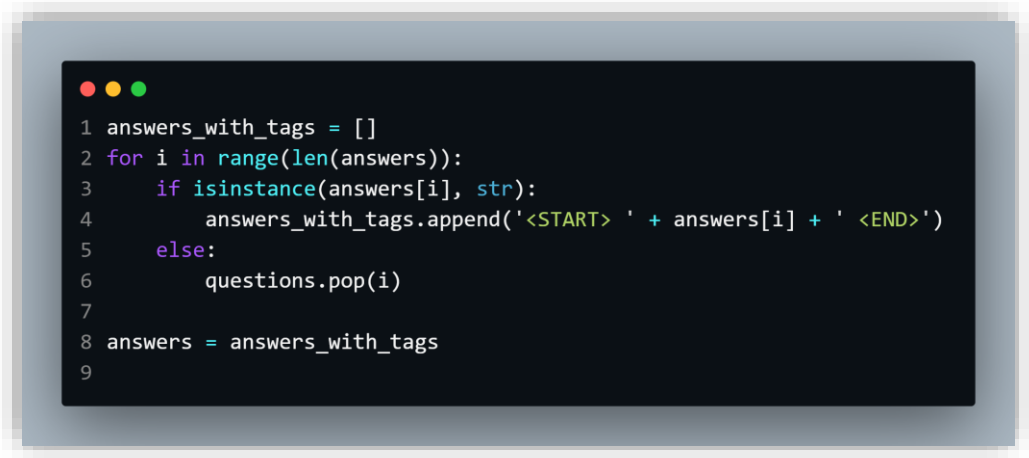
Procesamiento de Respuestas

Para que el modelo pueda responder en base al input que ingrese el usuario, es necesario realizar el siguiente procedimiento:

- Se debe de asignar etiquetas **<START> y <END>**

- Cada respuesta debe de contener una etiqueta **<START>** al inicio **y <END>** al final, los cuales indican los límites que debe de tener cada secuencia y ayudan al modelo a aprender cuando comenzar u cuando terminar una respuesta.

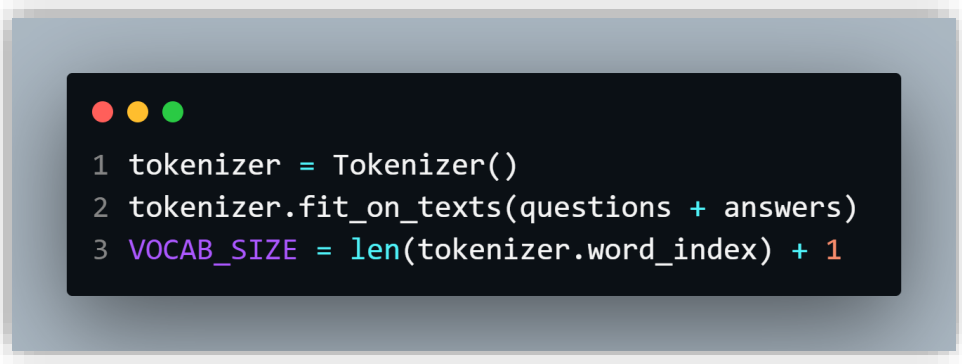
- Si una respuesta no es una cadena, esta será considerada invalida y el modelo recurrirá a eliminarla junto a su pregunta asociada.
- Una vez realizado este proceso, las respuestas están preparadas para poder ser tokenizadas y utilizadas en el decoder.



```
answers_with_tags = []
for i in range(len(answers)):
    if isinstance(answers[i], str):
        answers_with_tags.append('<START> ' + answers[i] + ' <END>')
    else:
        questions.pop(i)

answers = answers_with_tags

```

Tokenización

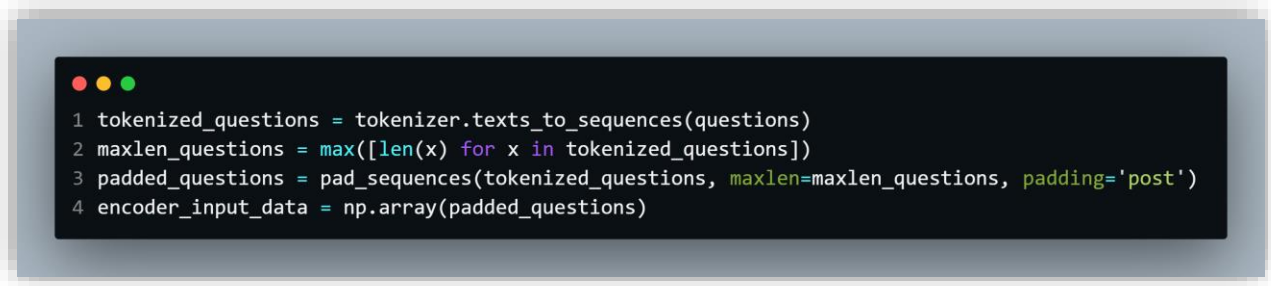
Es necesario poder utilizar un *Tokenizer*, el cual se encarga de poder generar un vocabulario basado en la frecuencia de las palabras que se utilizan en *questions* y *answer*. Para obtener dicho resultado es necesario realizar una construcción del vocabulario, para ello se utiliza el método *tokenizer.fit_on_texts*, el cual se encarga de poder analizar el texto ya asignarle un índice a cada palabra. Las palabras más frecuentes reciben índices más bajos. Además, se incluyen una índice adición, denominado *VOCAB_SIZE*, el cual es un índice adicional para representar el padding.



```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(questions + answers)
VOCAB_SIZE = len(tokenizer.word_index) + 1
```

Secuencias y Padding

Dado que ya fue realizada una tokenizacion, cada palabra en las preguntas se conviente en su incice correspondiente. Para realizar el padding, se ajusta a las secuencias una longitud uniforme, mediante la variable *maxlen_questions* y *padding='post'* añade ceros al final de las secuencias más cortas. Los datos finales se almacenan en una matriz denominada *encoder_input_data* donde cada fila representa una pregunta tokenizada y rellenada.



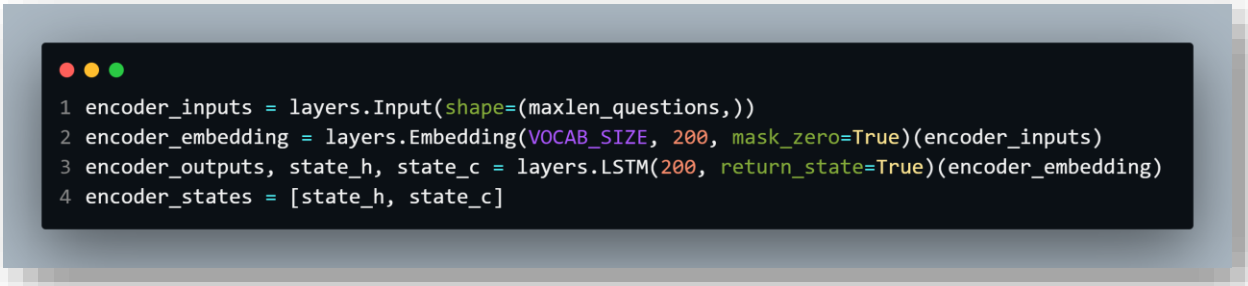
```
tokenized_questions = tokenizer.texts_to_sequences(questions)
maxlen_questions = max([len(x) for x in tokenized_questions])
padded_questions = pad_sequences(tokenized_questions, maxlen=maxlen_questions, padding='post')
encoder_input_data = np.array(padded_questions)
```

Construcción del modelo

Para realizar la construcción del modelo, es necesario dividirlo en dos etapas, *encoder* y *decoder*. Cuyo funcionamiento se desglosa de la siguiente manera:

- **Encoder**
  - Se realiza una capa de entrada *input*, la cual especifica la forma de entrada para las preguntas *maxlen_questions*.
  - La capa de embedding convierte los índices de palabras en vectores densos de tamaño de 200.

- LSTM se encarga de procesar cada una de las secuencias y las devuelve de la siguiente manera:
  - Encoder_outputs: Es la representación codificada de la secuencia completada.
  - *state_h* y *state_c* representa los estados finales del LSTM, utilizados como inicialización del modelo.



```
encoder_inputs = layers.Input(shape=(maxlen_questions,))
encoder_embedding = layers.Embedding(VOCAB_SIZE, 200, mask_zero=True)(encoder_inputs)
encoder_outputs, state_h, state_c = layers.LSTM(200, return_state=True)(encoder_embedding)
encoder_states = [state_h, state_c]
```

- **Decoder**
  - Se realiza una capa de entrada *input* que especifica la forma de entada para las respuestas *maxlen_answers.*
  - LSTM se encarga de generar las secuencias de salida usando los estados iniciales del encoder *encoder_states*.
  - Se genera una capa densa, la cual se encarga de calcular las probabilidades de cada palabra en el vocabulario utilizando *softmax*.

```
decoder_inputs = layers.Input(shape=(maxlen_answers,))
decoder_embedding = layers.Embedding(VOCAB_SIZE, 200, mask_zero=True)(decoder_inputs)
decoder_lstm = layers.LSTM(200, return_state=True, return_sequences=True)
decoder_outputs, _, _ = decoder_lstm(decoder_embedding, initial_state=encoder_states)
decoder_dense = layers.Dense(VOCAB_SIZE, activation=activations.softmax)
output = decoder_dense(decoder_outputs)

```

Entrenamiento del modelo del modelo

Para el correcto entrenamiento del modelo, es necesario realizar la siguiente categorización

- Entradas y Salidas del entrenamiento
    - encoder_input_data: Preguntas preprocesadas.
    - decoder_input_data: Respuestas preprocesadas con <**START**>.
    - decoder_output_data: Respuestas en formato one-hot, excluyendo <**START**>.
- Parámetros
    - batch_size=32: Número de muestras procesadas simultáneamente.
    - epochs=150: Iteraciones completas sobre los datos.

```
model.fit([encoder_input_data, decoder_input_data], decoder_output_data, batch_size=32, epochs=150)
```

Guardado del modelo y de los metadatos

- Se guarda el modelo como *s2s_model.keras*.
- Se persisten metadatos críticos (tokenizer, longitudes máximas) en un archivo .pkl.

```
model.save("s2s_model.keras")

metadata = {
    "tokenizer": tokenizer,
    "maxlen_questions": maxlen_questions,
    "maxlen_answers": maxlen_answers
}

with open('tokenizer.pkl', 'wb') as file:
    pickle.dump(metadata, file)

```

Fase de entrenamiento para la IA mediante en Intents

Procesamiento de la Entrada

Para que el modelo sea entrenado, es necesario convertir un texto de entrada en un vector binario (array de ceros y unos) según un vocabulario predefinido, para ello es necesario realizar dicha conversión y se usaran los siguientes elementos:

- input.toLowerCase().split(' '): Convierte el texto a minúsculas y lo divide en palabras usando los espacios como delimitadores.
- new Array(vocabulary.length).fill(0): Crea un vector del mismo tamaño que el vocabulario, inicializado con ceros.
- words.forEach(...): Este método se encarga de iterar cada palabra sobre el texto; si la palabra esta en el vocabulario, marca el índice correspondiente con un 1; si la palabra no está en el vocabulario, imprime una advertencia.

```
function preprocessInput(input: string, vocabulary: string[]): number[] {
  const words = input.toLowerCase().split(' ');
  const vector = new Array(vocabulary.length).fill(0);

  words.forEach((word) => {
    const index = vocabulary.indexOf(word);
    if (index !== -1) {
      vector[index] = 1;
    } else {
      console.warn(`Palabra desconocida: ${word}`);
    }
  });

  return vector;
}
```

Seleccionar respuesta aleatoria

Se procede a seleccionar una respuesta aleatoria en base a una lista de plausibles respuestas. Al proporcionar respuestas variadas, el chatbot parece menos predecible, lo que mejora la experiencia del usuario y permite que cada intent tenga múltiples respuestas predefinidas, lo que hace que el sistema sea más adaptable y versátil.:

```
function getRandomResponse(responses: string[]): string {
  const randomIndex = Math.floor(Math.random() * responses.length);
  return responses[randomIndex];
}
```

Generar Vocabulario

Se procede a crear un vocabulario único a partir de los datos de entrada para definir las palabras conocidas que el modelo puede interpretar. El vocabulario actúa como una referencia común para vectorizar las frases de entrada. Sin un vocabulario único, las palabras no se mapearían correctamente a índices consistentes. El vocabulario refleja las palabras más relevantes para la tarea, permitiendo al modelo enfocarse en datos específicos.

```
function generateVocabulary(data: { input: string; output: string[] }[]): string[] {
  const allWords = data
    .map((item) => item.input.toLowerCase().split(' '))
    .reduce((acc, words) => acc.concat(words), []);
  return Array.from(new Set(allWords));
}
```

Creación del modelo

Para definir la arquitectura de una red neuronal que clasifica entradas textuales (representadas como vectores) en una de varias categorías (intents). Se debe de redistribuir de la siguiente manera:

- Capa de Entrada - inputShape: [vocabularyLength]: Asegura que cada entrada tiene el mismo tamaño que el vocabulario. Cada vector representa la presencia/ausencia de palabras.
- Capas Ocultas – *units: 16 y 8*: Estas capas realizan transformaciones no lineales, capturando patrones complejos en los datos.
- Capas de salida:
  - units: outputLength: Representa el número de intents posibles.
  - activation: 'softmax': Genera probabilidades para cada intent. El modelo predice el intent con la probabilidad más alta.
- Compilación del modelo:
  - optimizer: 'adam': Algoritmo eficiente para ajustar los pesos del modelo.
  - loss: 'categoricalCrossentropy': Calcula qué tan diferente es la predicción del modelo respecto al objetivo.

```typescript
function getRandomResponse(responses: string[]): string {
  const randomIndex = Math.floor(Math.random() * responses.length);
  return responses[randomIndex];
}
function generateVocabulary(data: { input: string; output: string[] }[]): string[] {
  const allWords = data
    .map((item) => item.input.toLowerCase().split(' '))
    .reduce((acc, words) => acc.concat(words), []);
  return Array.from(new Set(allWords));
}
```

Entrenamiento del modelo

Entrenar el modelo para que aprenda a asociar entradas (frases vectorizadas) con salidas (intents). Las entradas (inputs) son vectores que representan frases de entrada y las salidas (labels) son vectores one-hot que representan los intents. Es necesario que al modelo se le asigne un modelo supervisado ajusta sus pesos internos para minimizar la diferencia entre sus predicciones y los valores esperados. El modelo es almacenado y entrenado permite su reutilización para predicciones futuras sin necesidad de volver a entrenarlo.

```typescript
async function trainModel(): Promise<void> {
  const vocabulary = generateVocabulary(trainingData);
  const model = createModel(vocabulary.length, trainingData.length);

  const inputs = trainingData.map((data) => preprocessInput(data.input, vocabulary));
  const labels = trainingData.map((_, i) => {
    const labelVector = new Array(trainingData.length).fill(0);
    labelVector[i] = 1;
    return labelVector;
  });

  const xs = tf.tensor2d(inputs);
  const ys = tf.tensor2d(labels);

  await model.fit(xs, ys, {
    epochs: 200,
    batchSize: 15,
    shuffle: true,
  });

  await model.save('localstorage://chatbot-model');
  console.log('Modelo entrenado y guardado.');
}
```

Predecir Respuesta

Una vez que el modelo haya sido entrenado para poder clasificar una nueva entrada y seleccionar una respuesta apropiada, para ello se recupera el modelo previamente entrenado y guardado. Para poder llegar a predecir el intent, se transforma la frase en un vector y calcula la probabilidad de cada intent y selecciona el intent con la mayor probabilidad. Para que la respuesta se pueda generar, se mapea intent predicho a un conjunto de posibles respuestas y selecciona una de ellas.

```typescript
async function predictResponse(input: string): Promise<string> {
  const vocabulary = generateVocabulary(trainingData);
  const model = await tf.loadLayersModel('localstorage://chatbot-model');

  const processedInput = preprocessInput(input, vocabulary);
  const tensorInput = tf.tensor2d([processedInput], [1, vocabulary.length]);

  const prediction = model.predict(tensorInput) as tf.Tensor;
  const predictedIndex = prediction.argMax(-1).dataSync()[0];

  const possibleResponses = trainingData[predictedIndex].output;
  return getRandomResponse(possibleResponses);
}
```

Tecnologías Utilizadas

- **Frontend:** Angular 12.x utilizando TypeScript
- **Desarrollo de Modelo IA:** Python 3.12.x

Integración de Librerías

- Numpy: Es una biblioteca fundamental para cálculos numéricos en Python, proporcionando herramientas eficientes para trabajar con matrices, tensores y operaciones matemáticas de bajo nivel. Durante el entrenamiento del modelo, las secuencias de entrada (preguntas) y las salidas (respuestas) se representan como matrices numéricas. Numpy permite transformar las listas tokenizadas en arreglos multidimensionales listos para el procesamiento por TensorFlow.

- Pandas: Es una biblioteca de análisis de datos diseñada para manipular y explorar estructuras tabulares o series de datos. Durante la etapa de preprocesamiento, Pandas se utiliza para estructurar y validar preguntas y respuestas extraídas de los archivos YAML. Permite verificar la consistencia de los datos antes de tokenizarlos.
- Tenserflow: Es un framework de aprendizaje automático ampliamente utilizado para construir, entrenar y desplegar modelos de deep learning. Este proporciona las capas Embedding, LSTM y Dense que conforman el modelo Seq2Seq, cuyas capas LSTM manejan el aprendizaje de patrones secuenciales, mientras que la capa densa con activación softmax genera predicciones. Esto nos permite compilar el modelo con el optimizador RMSprop y la pérdida categórica cruzada, garantizando una convergencia efectiva.
- Pickle: Es una biblioteca de serialización que convierte objetos de Python en flujos de bytes para almacenarlos o transmitirlos. El tokenizer, junto con el tamaño máximo de las secuencias (maxlen_questions y maxlen_answers), se guarda en un archivo para reutilizarse durante la inferencia. Lo cual se utiliza para deserializar los objetos almacenados previamente, facilitando el despliegue del modelo sin necesidad de reentrenarlo.
- PyYAML: Es una biblioteca para leer y escribir archivos en formato YAML, ampliamente utilizado para configurar datos estructurados. Se usa para cargar conversaciones de archivos YAML. Cada archivo contiene preguntas y respuestas en un formato estructurado, que se procesa para generar los datos de entrenamiento. Esto nos permite procesar archivos con diferentes estructuras dentro de una misma carpeta, garantizando que todos los datos relevantes sean incorporados al modelo.
- Gensim: Es una biblioteca especializada en procesamiento de lenguaje natural y modelado semántico, particularmente útil para análisis y representación vectorial de vocabularios. Complementa al tokenizer proporcionando herramientas para crear y explorar el vocabulario derivado de las preguntas y respuestas procesadas. Este facilita la tokenización avanzada y la limpieza del texto, asegurando que las palabras clave sean representadas correctamente en el modelo.

BIBLIOGRAFÍA

- Medium (2024), *Recurrent Neural Networks (RNN) from Basic to Advanced* *https://medium-com.translate.goog/@sachinsoni600517/recurrent-neural-networks-rnn-from-basic-to-advanced-1da22aafa009?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc&_x_tr_hist=true*
- Medium (2020), *Understanding Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM)*. https://medium.com/analytics-vidhya/undestanding-recurrent-neural-network-rnn-and-long-short-term-memory-lstm-30bc1221e80d
- Medium (2024*), Recurrent Neural Networks (RNN) from Basic to Advanced. https://medium.com/@sachinsoni600517/recurrent-neural-networks-rnn-from-basic-to-advanced-1da22aafa009*
- Medium (2023), *Introduction to Long Short-Term Memory (LSTM)*. https://medium.com/analytics-vidhya/introduction-to-long-short-term-memory-lstm-a8052cd0d4cd
- Medium (2021), *Long Short-Term Memory Networks*. https://medium.com/analytics-vidhya/long-short-term-memory-networks-23119598b66b
- Medium (2024), *Understanding Long Short-Term Memory (LSTM) in Deep Learning: A Comprehensive Guide with Sample Code*. https://medium.com/@hkabhi916/understanding-long-short-term-memory-lstm-in-deep-learning-a-comprehensive-guide-with-sample-72d884be4470
- Medium (2024), *Sequence-to-Sequence Models*. https://medium.com/@calin.sandu/sequence-to-sequence-models-603920ce9e96#:~:text=Seq2Seq%20models%20have%20been%20instrumental,text%20summarization%2C%20and%20speech%20recognition.&text=The%20encoder%20in%20a%20Seq2Seq,vector%20or%20a%20hidden%20state.
- Medium (2021), *Encoder-Decoder Seq2Seq Models, Clearly Explained*. https://medium.com/analytics-vidhya/encoder-decoder-seq2seq-models-clearly-explained-c34186fbf49b

- Medium. (2023). *Understanding Sequence-to-Sequence (Seq2Seq) Models and their Significance*. https://medium.com/aimonks/understanding-sequence-to-sequence-seq2seq-models-and-their-significance-d2f0fd5f6f7f