UNAHUR
UNIVERSIDAD NACIONAL DE HURLINGHAM

Trabajo Práctico 3

Aprendizaje Automático Avanzado

Cisnero Matias, Seivane Nicolás, Serafini Franco

17 de Noviembre de 2025

Este trabajo práctico tiene como objetivo analizar e implementar dos variantes arquitectónicas de Transformadores para traducción automática español-inglés:
Implementación P: xxx
Implementación A: xxx

Contexto

Los Transformadores representan un gran avance en el campo del Procesamiento de Lenguaje Natural (PLN). En este contexto los mismos constituyen el estado del arte actual para el aprendizaje de Modelos de Grandes Lenguajes (LLM).
Para realizar traducción automática, los Transformadores implementan el modelo de secuencia a secuencia (seq2seq), que es capaz de obtener una secuencia de salida dada una secuencia de entrada, independientemente de la longitud de ambas. Estos sistemas son entradnos a partir de pares de oraciones, una de ellas en el lenguaje fuente y la otra en el lenguaje objetivo.

Arquitectura General

El traductor esta compuesto por dos Transformadores:

- Codificador: Procesa la secuencia de entrada y genera representaciones contextuales enriquecidas(prof).
- Decodificador: Genera la secuencia de salida token por token, utilizando la información proporcionada por el Codificador.

Durante el entrenamiento se busca maximizar la probabilidad:

$$P(y_1, y_2, \ldots, y_m \mid x_1, x_2, \ldots, x_n)$$

donde x es la oración a traducir e y es la oración traducida.

Embeddings y Codificación Posicional

Cada token se representa mediante:

- **Matriz E**: $E \in \mathbb{R}^{|V| \times d}$ donde cada fila contiene la representacion contextual de una palabra.
- **Codificacion One-Hot**: Se multiplica por E para obtener el embedding correspondiente.

Debido a que la posición de las representaciones contextuales en una entrada aporta información acerca de la relación entre ellas, se le suma a cada token un vector de la misma dimensión d, cuyo valor depende de la posición del token en la secuencia de entrada. La función para obtener dichos vectores es:

$$vp(p, j) = \begin{cases} \sin\left(\frac{p}{10000^{2j/d}}\right) & \text{si } j \text{ es par} \\ \cos\left(\frac{p}{10000^{(2j-1)/d}}\right) & \text{si } j \text{ es impar} \end{cases}$$

Mecanismo de Auto-atencion

Cada token adopta tres roles:

- **Consulta (Query)**: El token es transformado para poder ser comparado con otras palabras de la entrada.
- **Clave (Key)**: El token es transformado para poder ser comparado con la palabra en foco.
- **Valor (Value)**: El token es transformado para poder calcular el valor final de la representación de cada palabra.

Así, las representaciones de las palabras según sus roles son:

$$q_i = x_i W^Q,$$
$$k_i = x_i W^K,$$
$$v_i = x_i W^V$$

Mecanismo de Auto-Atención

Calculo de la nueva representación contextual a_i

La nueva representación contextual a_i para cada token se calcula como:

$$a_i = \sum_{j \leq i} \alpha_{i,j} v_j = \sum_{j \leq i} \text{softmax}\left(\frac{q_i k_j}{\sqrt{d_k}}\right) v_j$$

Atención Múltiple

Múltiples cabezales permiten al modelo capturar diferentes tipos de relaciones lingüísticas como relaciones sintácticas o semánticas.

Cada cabezal k tiene sus propias matrices de pesos:

$$Q_k = XW_k^Q, \quad K_k = XW_k^K, \quad V_k = XW_k^V$$

Y cada cabezal calcula:

$$Y_k = \text{softmax}\left(\frac{Q_k K_k^t}{\sqrt{d_k}}\right) V_k$$

Mecanismo de Auto-Atencion

Concatenación:

$$A = (Y_1 +' Y_2 +' \ldots +' Y_h)W^O$$

donde $Y_k \in \mathbb{R}^{N \times d_v}$ es la salida del cabezal k, la concatenación produce una matriz de $N \times hd_v$, y $W^O \in \mathbb{R}^{hd_v \times d}$ deja la salida de dimension Nxd.

Capa de PMC y Normalización

Capa de N Perceptrones Multicapa (PMC)

Cada bloque contiene una capa de N PMCs con:

- Dimensión d_{ff}.
- Función de activación GELU.
- Los pesos son compartidos entre todos los tokens.

Normalizacion por Capa:

- Calcula la media y la desviación sobre los elementos del vector de representación de cada token.
- Aplica la transformacion lineal $\gamma x + \beta$ donde γ y β son parámetros aprendidos durante el entrenamiento.

Conexiones residuales:

- Son conexiones directas entre la entrada de cada capa y su salida correspondiente.
- Se aplican tanto a la capa de Auto-Atencion como a la de PMC.

Punto 2 - Conjunto de Datos Utilizados

Common Crawl

Es una organización sin fines de lucro que rastrea la web y ofrece un repositorio masivo y gratuito de datos de rastreo web para investigadores, empresas y público en general. Estos datos, que incluyen petabytes de páginas web, se almacenan en Amazon Web Services (AWS) y se actualizan periódicamente.

¡¡Problema!!

Las millones de páginas están guardadas en el lenguaje de marcado (HTML).

¿La solución? Hay varias bases de datos con estos HTML´s depurados, que luego pueden ser utilizados como *tokens* para diferentes modelos de lenguajes.

Normalización del Corpus ('a')

Archivo .txt

Se lee el archivo `.txt` con pares de oraciones separados por tabulador.

Normalización

- Conversión a minúsculas y normalización Unicode (`NFKC`).
- Inserción adecuada de espacios alrededor de signos de puntuación mediante expresiones regulares.
- Separación en inglés y español usando el carácter `\t`.

Las oraciones en ingles se marcan con tokens especiales:

`[start]` al inicio y `[end]` al final.

Se generan los pares normalizados `(esp, eng)` y se guardan en `text_pairs.pickle`.

Vectorización e Importancia de `pickle('a')`

Vectorización

- Se divide el corpus en **train** / **validación** / **test**.
- Se crean dos capas `TextVectorization` y se adaptan con el conjunto de entrenamiento.
- Los vectorizadores entrenados y los pares de datos se guardan con `pickle`.

¿Por qué es importante `pickle`?

- Permite guardar objetos complejos de Python (`TextVectorization`, vocabularios, listas, diccionarios).
- Evita recalcular o readaptar los vectorizadores: se mantienen **exactamente los mismos tokens y el mismo índice de cada palabra**.

Limpieza y Preparación del Corpus ('p')

Archivo .txt

Se carga el archivo `.txt` y se separa en pares `(inglés, español)` usando el tabulador.

Normalización

- Normalización Unicode (`NFD`) y eliminación de caracteres no ASCII.
- Conversión a minúsculas y Remoción de puntuación mediante tabla de traducción.
- Eliminación de tokens con números y caracteres no imprimibles.
- Reconstrucción de la oración limpia como string.

El resultado final se almacena como matriz NumPy y se guarda en `ing-esp.pkl`.

Se agrega marcado con tokens especiales

- <SOS> al inicio y <EOS> al final.

Tokenización y Dataset ('p')

Se utilizan la clase `PrepareDataset` y `Tokenizer` de Keras para:

- Ajustar un tokenizador para el encoder (oraciones fuente).
- Ajustar otro tokenizador para el decoder (oraciones objetivo).
- Calcular longitudes máximas y tamaños de vocabulario.
- Convertir texto a secuencias enteras y aplicar `padding`.

- Se generan:
  - `trainX`: oraciones fuente codificadas.
  - `trainY`: oraciones destino codificadas.
  - `train_dataset`: batches de entrenamiento con `tf.data.Dataset`.

Funciones de Keras utilizadas (Implementación a)

TextVectorization

- Tokeniza, normaliza y convierte texto en secuencias de enteros.
- Aprende el vocabulario mediante `adapt()`.
- Permite definir:
  - tamaño del vocabulario,
  - longitud máxima de secuencia,
  - modo de salida (entero, multi-hot, TF-IDF).
- Se usa para construir:
  - `encoder_inputs` (español),
  - `decoder_inputs` y `targets` (inglés).

tf.data.Dataset

- Construye el pipeline eficiente de entrenamiento.
- Permite `batch`, `shuffle`, `prefetch` y `cache`.

No generan embeddings: sólo producen secuencias de índices

Funciones de Keras utilizadas (Implementación p)

Tokenizer()

- Construye un vocabulario a partir del texto.
- Convierte cada palabra en un entero único.
- No normaliza automáticamente: se usa texto ya limpiado.
- Se guardan dos tokenizadores:
  - uno para el encoder,
  - otro para el decoder.

texts_to_sequences()

- Transforma cada oración en una lista de índices enteros.
- Si una palabra no está en el vocabulario $\rightarrow$ índice `OOV`.

Funciones de Keras utilizadas (Implementación p)

pad_sequences()

- Ajusta todas las oraciones a una longitud fija.
- Agrega ceros al final (`padding='post'`).

tf.data.Dataset

- Crea un dataset por lotes para entrenamiento del transformer.

No se generan embeddings. Sólo secuencias de índices.

Comparación entre TextVectorization y Tokenizer

TextVectorization

- Incluye normalización integrada.
- Se adapta con `adapt()`.
- Permite modos de salida alternativos (TF-IDF, multi-hot).

Tokenizer()

- No normaliza: requiere limpieza manual previa.
- Muy usado en implementaciones clásicas de seq2seq.
- Permite guardar tokenizadores fácilmente con `pickle`.

Ambas producen secuencias enteras para alimentar una capa `Embedding`.

Embeddings en Transformers (Implementación Real)

¿Qué hace la capa `Embedding`?

- Convierte cada token entero en un vector d-dimensional.
- Los vectores se inicializan aleatoriamente.
- Son **parametrizables**: se ajustan durante el entrenamiento.
- Representan relaciones semánticas aprendidas por gradiente.

Diferencia con CBOW / Skip-gram

- CBOW/Skip-gram aprenden embeddings usando contextos.
- Los vectores representan co-ocurrencia de palabras reales.
- Son embeddings **preentrenados** basados en corpus grandes.
- La capa `Embedding` del Transformer:
  - NO usa contextos para aprender.
  - Aprende tareas específicas (p. ej. traducción).
  - Se entrena **junto con todo el modelo**.

Embedding Posicional Entrenable vs. Fijo

Embedding Posicional Entrenable (Implementación a)

- Se usa `tf.keras.layers.Embedding`.
- Inicializa vectores para cada posición.
- Los valores son parte del entrenamiento.
- Se suman vector a vector con los token embeddings.
- Modelo aprende relaciones posicionales óptimas.

Embedding Posicional Fijo (Implementación p)

- Usa matriz sinusoidal (**no entrenable**).
- Valores calculados: $\sin(k/n^{2i/d})$ y $\cos(k/n^{2i/d})$.
- Se suma a los token embeddings fijos.
- No se modifica durante entrenamiento.
- Permite al modelo razonar sobre distancias absolutas y relativas.

Implementación a: PositionalEmbedding (Entrenable)

¿Qué hace esta implementación?

- Crea embeddings de palabras **entrenables**.
- Crea embeddings posicionales fijos generados por seno y coseno.
- El método `call()`:
  - Convierte tokens en vectores densos.
  - Suma vectores posicionales precomputados.
- Todos los embeddings de palabras se ajustan con gradiente.

Implementación p: PositionEmbeddingFixedWeights (No Entrenable)

¿Qué hace esta implementación?

- Genera **embeddings de palabras** mediante sinusoidales.
- Usa `Embedding(..., trainable=False)`.
- Genera embeddings posicionales también fijos.

Limitaciones

- Embeddings de palabras NO representan semántica real.
- No aprenden durante entrenamiento.
- No se parecen a CBOW/Skip-gram ni a embeddings reales.

Punto 4 - Implementación p

En la implementación p el modelo se almacena de forma modular.
En el archivo p9.py se genera el archivo ing-esp.pkl, que contiene:

- Las oraciones limpias en inglés.
- Las oraciones limpias en español.

El tokenizador se guarda en el archivo p10.py mediante:

```
self.save_tokenizer(enc_tokenizer, 'enc')
self.save_tokenizer(dec_tokenizer, 'dec')
```

Lo cual produce los archivos:

- `enc_tokenizer.pkl`
- `dec_tokenizer.pkl`

Punto 4 - Implementación p

Pesos del modelo

La implementación **p** guarda los parámetros mediante TensorFlow Checkpoints, como se ve en el archivo p11.py:

```
    ckpt = train.Checkpoint(model=training_model,
optimizer=optimizador)
    ckpt_manager = train.CheckpointManager(ckpt, "./
checkpoints", max_to_keep=3)
```

Durante el entrenamiento, después de cada época se almacena:

```
    save_path = ckpt_manager.save()
```

Se guardan:

- Los pesos del codificador.
- Los pesos del decodificador.
- El estado del optimizador Adam.

Para restaurar un entrenamiento previo se utiliza:

Punto 5 - Implementación 'a'

En la implementación **'a'** el mecanismo de **multi-auto-atención** se implementa usando tf.keras.layers.MultiheadAttention de TensorFlow:

```
    def self_attention(input_shape, prefix="att", mask=
False, **kwargs):
        inputs = tf.keras.layers.Input(shape=input_shape,
dtype='float32',
                                       name=f"{prefix}_in1")
        attention = tf.keras.layers.MultiHeadAttention(name=
f"{prefix}_attn1", **kwargs)
        # ...
```

```
    def cross_attention(input_shape, context_shape, prefix
="att", **kwargs):
        context = tf.keras.layers.Input(shape=context_shape,
 dtype='float32',
                                       name=f"{prefix}_ctx2")
        inputs = tf.keras.layers.Input(shape=input_shape,
dtype='float32',
```

Punto 5 - Implementacion 'p'

En la implementación 'p' la capa de multi-auto-atención se implementa manualmente:

```
class MultiAutoAtencion(Layer):
def __init__(self, cabezales, d_k, d_v, d, **kwargs):
self.atencion = AutoAtencion()
self.cabezales = cabezales
self.d_k = d_k
self.d_v = d_v
self.d = d
self.W_q = Dense(d)
self.W_k = Dense(d)
self.W_v = Dense(d)
self.W_o = Dense(d)
```

Punto 6 - Implementación 'a' - Obtención del Vocabulario

En la implementación 'a' el vocabulario se obtiene usando `TextVectorization` de Keras:

```
eng_vectorizer = TextVectorization(
  max_tokens=vocab_size_en,
  output_mode="int",
  output_sequence_length=seq_length,
)

# Adaptacion con datos de entrenamiento
train_eng_texts = [pair[0] for pair in train_pairs]
eng_vectorizer.adapt(train_eng_texts)
```

Proceso:

- Se crea una capa `TextVectorization` con tamaño máximo de vocabulario
- La capa automáticamente construye el vocabulario
- Se guarda junto con el modelo en el archivo `vectorize.pickle`

Punto 6 - Implementación 'p' - Obtención del Vocabulario

En la implementación 'p' el vocabulario se obtiene usando `Tokenizer` de Keras:

```
def create_tokenizer(self, dataset):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(dataset)
  return tokenizer

# Uso para encoder y decoder
enc_tokenizer = self.create_tokenizer(train[:, 0])
dec_tokenizer = self.create_tokenizer(train[:, 1])
```