# **Capítulo 7. Procesamiento del lenguaje natural con RNNs**

## **Introducción**

**¿Cómo podemos saber cuándo una computadora ya cobró conciencia?**

Muchos de ustedes dirían que aplicando la prueba de Turing – muchos de ustedes estarían en lo incorrecto.

La prueba de Turing, o el juego de la imitación, fue propuesta por Alan Turing en 1950 para determinar la capacidad de una máquina de exhibir comportamiento inteligente indistinguible al de un humano. No sabemos si hay conciencia involucrada, pero si tu algoritmo se hace wey al entrevistador que está aplicando la prueba, pasó la prueba de Turing.

Para la prueba, la propuesta de Alan Turing era que un entrevistador sometiera a una computadora y a un humano a una conversación de chat en lenguaje natural.

La idea era que, durante el chat, el entrevistador no pudiera distinguir cual es el humano y cuál es la computadora.

Ahora, Turing podría haber escogido montón de diferentes facetas de la inteligencia humana para evaluar: la capacidad de poder reconocer un plátano en las fotos, jugar ajedrez, componer música de banda, o navegar un sendero escabroso.

Al solo enfocarse en texto, Turing limito bastante su prueba y por lo mismo ha sido bastante criticada y controversial. Cualquier programador puede codificar un set de reglas diseñadas específicamente para pasar la prueba de Turing, sin mostrar inteligencia natural.

Mucha gente, hombres y mujeres, han aceptado citas de Tinder de perfiles falsos que resultaron ser chatbots de investigadores en universidades – en mi opinión el chatbot no paso la prueba de Turing, sino que más bien los humanos que aceptaron la cita la reprobaron.

En temas más serios, el desarrollo de redes neuronales como GPT-3 ha permitido a las computadoras escribir con una elocuencia y naturalidad que muchos seres humanos no alcanzamos. Es más, es altamente probable que algún resumen de partido o reportaje sobre movimientos en la bolsa que hayas leído recientemente haya sido el producto de una red neuronal.

Así que… ¿Las computadoras ya pasaron la prueba de Turing? En el sentido estricto en el que Alan Turing planteo el examen inicial, técnicamente la respuesta es sí… con asterisco.

La respuesta de verdad se las vamos a dejar a los filósofos y nosotros nos enfocaremos en el grueso del capítulo – como podemos generar redes neuronales que procesen lenguaje natural y lo puedan interpretar, o crear correctamente.

Vamos a ver que hay 2 maneras de procesar lenguaje natural – la que históricamente ha dominado pero que ya está por terminar de caer en la obsolescencia (los RNNs) y la chica nueva de la cuadra, los Transformers.

Comenzaremos este capítulo creando un RNN básico que nos permitirá predecir el siguiente carácter en una frase. No palabra, carácter. Por el trabajo que realizan, estos se llaman, sorprendentemente, Character-RNNs.

Podremos crear un poco de texto original y veremos cómo podemos construir un Dataset de TensorFlow en una secuencia MUY larga. Si alguna vez has usado alguna de las sugerencias de autocompletar en el chat de tu celular, has usado una red neuronal de tipo Character – RNN o similar.

Para esta tarea conoceremos lo que es el RNN sin estado (o stateless RNN, en inglés), que es un RNN que aprende en porciones aleatorias del texto en cada iteración, sin informarse sobre el resto del texto.

Luego construiremos un RNN con Estado (stateful RNN) que preserva el estado oculto entre varias iteraciones de entrenamiento y continúa construyendo donde la anterior termino, permitiéndole aprender patrones más largos.

Una vez que hayamos terminado nuestro Character RNN, vamos a pasar a construir un RNN para llevar a cabo un “Análisis de Sentimiento”, en el cual leeremos reseñas de películas para saber si son positivas o negativas para la película en cuestión. Esta vez, nos fijaremos en las palabras, no en los caracteres, así que veremos las oraciones a nivel palabra.

Finalmente, pasaremos al ejercicio que nos tomara casi todo el capítulo – construir una arquitectura de tipo Codificador – Decodificador (Encoder/Decoder) para poder traducir textos de un idioma a otro. Esta actividad se le llama Traducción Neuronal de Máquina (Neural Machine Translation).

Una vez que vimos que nuestro Encoder-Decoder de RNN funciona, pasaremos el resto del capítulo hablando sobre porque es una basura que merecía ser abortada en la cuna. Ok, no tan grave, pero veremos que existen unas cosas llamadas “Mecanismos de Atención” que son mucho muy superiores a las arquitecturas tradicionales de Encoder Decoder.

Al final veremos que la atención es tan, pero tan poderosa, que los RNNs salen sobrando – mediante una capa nueva llamada Atención de Multi Cabeza veremos que se usó el concepto de atención para construir una arquitectura llamada Transnformer que es la sensación actualmente en aprendizaje profundo. Veremos si pasa de moda pronto o no.

Vamos Comenzando

## **Texto de Cervantes usando RNN**

En el 2015, Andrej Karpathy mostro como entrenar un RNN en un Blog llamado “The unreasonable effectiveness of Neural Networks” (la efectividad increíble de las redes neuronales recurrentes) que predice el siguiente carácter en una oración. Este Char-RNN como le estaremos diciendo de ahora en adelante puede ser usado para generar texto nuevo, un carácter a la vez.

El ejemplo que se usa de manera clásica es de Shakespeare, en inglés obviamente. Lo que hicieron fue agarrar todas las obras del famoso autor, pasarlas a un archivo .txt, alimentárselo a la máquina y usarlo para entrenar el Char-RNN.

Ahora, si tú le das un poquito de entrada, la máquina te autocompleta todo con una oración que está escrita en el estilo de Shakespeare.

Chéquense este ejemplo:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Para ustedes que no son angloparlantes nativos, ¿me la compran que salió directo de la obra de Enrique V?

Nel, salió de la computadora. No es una obra maestra y cualquiera que estudie letras inglesas te diría que esta chafa la redacción, pero el modelo puedo aprender palabras, gramática, puntuación apropiada y más, solo aprendiendo a predecir el siguiente carácter en una oración.

Como ya se estarán imaginando, vamos a construir nuestro propio Char-RNN.

Lo más importante, no lo haremos en inglés (como en todas las clases de redes neuronales), sino en español.

El máximo exponente de la lengua española es Miguel de Cervantes, y vamos a usar su obra para entrenar nuestro propio Char-RNN. Muchachos, diferencia principal, Shakespeare era un escritor de teatro y sus obras son cortitas. Hamlet es un panfletito que se lee en 1hr y media.

Cervantes por otro lado era novelista y sus obras pueden usarse como objetos para defensa personal, nada más chéquense el tamaño de 2 tomos del Quijote. Con ese si matas un cristiano.

Así que, no entrenaremos la red con las obras completas de Cervantes, solo con el quijote.

### **Creando el set de Entrenamiento**

Como primera instancia será necesario preparar nuestro libro de trabajo para esto hay que importar las librerías necesarias.

#Importaciones comunes

import numpy as np

import pandas as pd

#Importar TF y Keras

import tensorflow as tf

#Preparar Matplotlib

import matplotlib as plt

#Semillas a 42

np.random.seed(42)

tf.random.set\_seed(42)

#Print de versión

print(tf.\_\_version\_\_)

Vamos ahora descargando toda la obra de Cervantes, una vez la tengas es necesario colocarlo en la misma carpeta en donde tienes tu cuaderno de Jupyter.

with open("quijote.txt") as f:

quijote\_text = f.read()

Vamos imprimiendo los primeros 100 caracteres del texto:

print(quijote\_text[:100])



Después visualizaremos todos los caracteres del texto mediante el siguiente código:

"".join(sorted(set(quijote\_text.lower())))



Ahora, vamos a procesar cada carácter como un entero. Para eso utilizaremos la clase de Keras llamada **TextVectorization**. Con esto mapearemos el texto del Quijote a caracteres. Cada uno de estos caracteres será cambiado a un número. El carácter que más se repita en el texto, tendrá el valor entero más bajo.

La separación del texto será a nivel “character” partiendo de que los caracteres son minúsculas. Con el siguiente código estaremos creando la capa, pero aun no ha sido aplicado.

text\_vec\_layer = tf.keras.layers.TextVectorization(split = "character", standardize = "lower")

Ahora sí, le pasaremos nuestro texto a la capa para que haga su trabajo.

text\_vec\_layer.adapt([quijote\_text])

Una vez aplicado el método ahora será necesario guardarlo en alguna variable, para esto colocamos:

encoded = text\_vec\_layer([quijote\_text])[0]

Para poder visualizar lo realizado en los pasos anteriores, se pueden llamar los primero cien caracteres del array del encoded, estos números representan las palabras del texto del Quijote:

encoded[:100]

Imagen en blanco y negro de un teclado

Descripción generada automáticamente con confianza media

Para esta función el cero y el uno son utilizados como relleno (padding), es por eso por lo que no se ven en nuestra matriz como uno de los caracteres más utilizados. En este caso por esto sería el número 2.

Para deshacernos de estos dos tokens usaremos:

encoded -= 2

Continuaremos extrayendo el total de caracteres distintos en el texto, sin embargo, será necesario restarle dos por el mismo tema del padding, esto mediante:

n\_tokens = text\_vec\_layer.vocabulary\_size()-2

n\_tokens



El número que buscamos de caracteres es este 66 que nos apareció. Para saber el total de caracteres en todo el texto podemos usar:

dataset\_size = len(encoded)

dataset\_size



### **Ventanas para nuestro Dataset Secuencial**

Ahorita en este instante, nuestro set de entrenamiento consiste en una sola secuencia de más de un millón de caracteres, así que no podemos entrenar la red neuronal directamente – por la naturaleza de los RNNs acabaríamos con una red neuronal recurrente que itera sobre si misma varios millones de veces y eso sería un nivel de profundidad para el que no estamos preparados para resolver.

En vez de eso, vamos a aprovechar un método llamado window() o ventana en español para convertir esta secuencia de caracteres en varias ventanas mucho más pequeñas de texto. Cada instancia en el dataset va a ser un substring bastante corto de todo el texto, y nuestra RNN solo tendrá que iterar sobre estas ventanas de 100 caracteres. Entonces no tenemos el problema de acabar con una sola instancia de 1,000,000 de capas o más.

Este proceso de separar el dataset original en pedazos más chicos y alimentarlos de uno por uno al RNN se llama Backpropagacion truncada a través del tiempo. Vamos viendo como se hace con código:

def to\_dataset(sequence, length, shuffle = False, seed = None, batch\_size = 32):

ds = tf.data.Dataset.from\_tensor\_slices(sequence)

ds = ds.window(length + 1, shift = 1, drop\_remainder = True)

ds = ds.flat\_map(lambda window\_ds: window\_ds.batch(length+1))

if shuffle:

ds = ds.shuffle(100\_000, seed = seed)

ds = ds.batch(batch\_size)

return ds.map(lambda window: (window[:,:-1],window[:,1:])).prefetch(1)

Ahora vamos a dividir el dataset en train, valid y test usando la función que definimos anteriormente.

length = 100

train\_set = to\_dataset(encoded[:1\_000\_000], length = length, shuffle = True, seed = 42 )

valid\_set = to\_dataset(encoded[1\_000\_000:1\_060\_000], length = length)

test\_set = to\_dataset(encoded[1\_060\_000:], length=length)

Continuaremos con la creación de nuestro modelo, este será sencillo para el entrenamiento. Este modelo es basado en la arquitectura “Transformer” que veremos después en este capitulo a profundidad. Este modelo está basado en unas tres capas densas, un embedding, un GRU y al último una capa densa con activación softmax. El número de neuronas serán los 66 caracteres del texto, esto tiene lógica ya que al final de cuentas es un modelo de clasificación.

model = tf.keras.Sequential([

tf.keras.layers.Embedding(input\_dim = n\_tokens, output\_dim = 16),

tf.keras.layers.GRU(128, return\_sequences = True),

tf.keras.layers.Dense(n\_tokens, activation = "softmax")

])

Continuaremos con la compilación:

model.compile(loss="sparse\_categorical\_crossentropy", optimizer="nadam",

metrics=["accuracy"])

Debido a la complejidad de computo del modelo, necesitaremos check points:

model\_ckpt = tf.keras.callbacks.ModelCheckpoint(

"quijote\_modelo", monitor="val\_accuracy", save\_best\_only=True)

### **Construir y entrenar el modelo de Char-RNN**

El entrenamiento del modelo será una tarea ardua para nuestras computadoras, podría tardar hasta 10 min por época y si su computadora no es muy buena esto se alargará mucho más. Vamos ejecutando el modelo:

history = model.fit(train\_set, validation\_data = valid\_set, epochs = 10, callbacks = [model\_ckpt])

Si es que se dificulta el entrenamiento del modelo demasiado podrás usar el modelo cargado que dejaremos.

model = tf.keras.models.load\_model("quijote\_modelo")

Para visualizar los pesos de la red entrenada usaremos:

print(model.layers[0].weights)

Ahora cargaremos el Código para armar el modelo final:

quijote\_modelo = tf.keras.Sequential([

text\_vec\_layer,

tf.keras.layers.Lambda(lambda X: X-2),

model

])

### **Usando el Char-RNN generando texto falso**

Y ahora sí, vamos usando el modelo para predecir el siguiente carácter en un texto escrito por Cervantes. Usaremos una temperatura baja de 1, una temperatura baja hará que el modelo tienda a tomar al azar el siguiente carácter obteniendo un texto sin sentido alguno. Esto significa que el texto tomara las probabilidades más altas para el siguiente carácter como salida. Noten que sumamos dos por el padding que eliminamos anteriormente.

def next\_char(text, temperature = 1):

y\_proba = quijote\_modelo.predict([text])[0,-1:]

rescaled\_logits = tf.math.log(y\_proba)/temperature

char\_id = tf.random.categorical(rescaled\_logits, num\_samples = 1)[0,0]

return text\_vec\_layer.get\_vocabulary()[char\_id+2]

Ahora crearemos una función para crear la secuencia completa, esta función se alimenta con el texto, el número de caracteres y la temperatura:

def extend\_text(text, n\_chars = 50, temperature = 1):

for \_ in range(n\_chars):

text += next\_char(text, temperature)

return text

Podemos probar nuestra función con una de las frases más populares del Quijote “La virtud más es perseguida de los malos". En este caso esperamos nos complete la oración con “que amada de los buenos”. Noten que la temperatura es muy baja y es por eso que se repiten las palabras.

print(extend\_text("La virtud más es perseguida de los malos", temperature=0.01))



Ahora probaremos con una temperatura un poco más alta, noten que al subir la temperatura ya no se repiten datos:

print(extend\_text("La virtud más es perseguida de los malos", temperature = 1))



Luego con una temperatura super alta:

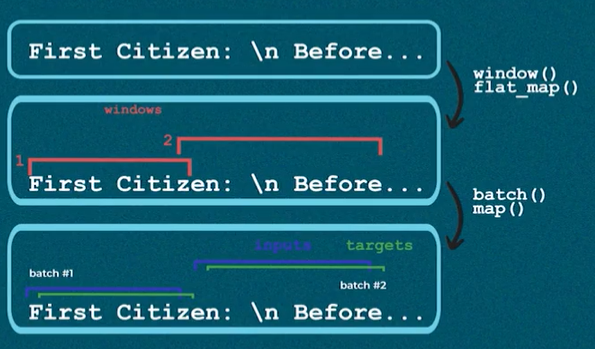
print(extend\_text("La virtud más es perseguida de los malos", temperature=100))



### **Stateful RNN**

Hasta ahora solo hemos usado Stateless RNNs, en cada iteración de entrenamiento el modelo comienza con un estado oculto lleno de ceros, y luego actualiza este estado en cada paso de tiempo, y después del último paso de tiempo, tira a la fregada el estado oculto porque ya no lo necesita.

Ya estarán pensando que tal vez podríamos reciclar este estado oculto y usarlo para el siguiente bache de entrenamiento. De esta manera, el modelo puede aprender patrones de largo plazo a pesar de que solo se propaga de regreso en secuencias cortas, esto se llama Stateful RNN.



Antes de empezar no quiero que se me vayan con la idea de que los Stateful RNNs son el santo grial. Los stateful RNNs solo tiene sentido usarlos donde la secuencia correspondiente dejo el asunto. Así que lo primero que necesitamos es usar secuencias de entrada que sean secuenciales y no se traslapen (así que no barajeamos nada).

Comenzamos creando el dataset, y luego hacer ventanas que comiencen exactamente donde termino el periodo de entrenamiento de la última ventana. Es decir, se acabaron las ventas que van de 0-100, 1-101, 2-102 y así. Ahora vamos de 0-100, 100-200, etc. ¿Por qué estamos traslapando de cualquier manera el ultimo carácter de cada ventana?

Porque ese va a ser nuestro dato de salida, como no se usó para entrenar el set anterior, podemos usarlo para entrenar el set que sigue.

### **Stateful RNN – Código**

Comenzamos creando una función que nos permita usar el modelo de stateful RNN, esta función es muy parecida a la pasada que vimos, el cambio que hacemos es en el valor de shift, esto para lograr tener las ventanas (windows) consecutivas:

def to\_dataset\_for\_stateful\_rnn(sequence,length):

ds = tf.data.Dataset.from\_tensor\_slices(sequence)

ds = ds.window(length + 1, shift=length, drop\_remainder=True)

ds = ds.flat\_map(lambda window: window.batch(length + 1)).batch(1)

return ds.map(lambda window: (window[:, :-1], window[:, 1:])).prefetch(1)

Separamos de nuevo en validación, entrenamiento y prueba:

stateful\_train\_set = to\_dataset\_for\_stateful\_rnn(encoded[1\_000\_000:], length)

stateful\_valid\_set = to\_dataset\_for\_stateful\_rnn(encoded[1\_000\_000:1\_600\_000], length)

stateful\_test\_set = to\_dataset\_for\_stateful\_rnn(encoded[1\_060\_000:], length)

Después tendremos que aplicar el modelo secuencial en Keras, muy parecido al paso salvo por dos cambios: El embedding y el parámetro de GRU stateful=True.

model = tf.keras.Sequential([

    tf.keras.layers.Embedding(input\_dim=n\_tokens, output\_dim=16,

                              batch\_input\_shape=[1, None]),

    tf.keras.layers.GRU(128, return\_sequences=True, stateful=True),

    tf.keras.layers.Dense(n\_tokens, activation="softmax")

])

Ahora vamos a resetear los estados, para esto armaremos un clase llamada “RestStatesCallbacks”.

class ResetStatesCallback(tf.keras.callbacks.Callback):

    def on\_epoch\_begin(self, epoch, logs):

        self.model.reset\_states()

Ahora un Checkpoint para guardar el progreso debido a la complejidad del modelo:

model\_ckpt = tf.keras.callbacks.ModelCheckpoint(

    "qujote\_stateful\_modelo",

    monitor="val\_accuracy",

    save\_best\_only=True)

Compilamos y ejecutamos el modelo:

history = model.fit(stateful\_train\_set, validation\_data=stateful\_valid\_set,

                    epochs=10, callbacks=[ResetStatesCallback(), model\_ckpt])

Y eso es todo ya quedo nuestro modelo, obviamente hay que entrenar y probar, pero eso quedara de tarea para ustedes.

## **Análisis de Sentimiento**

Una de las tareas más básicas en procesamiento de lenguaje natural es el análisis del sentimiento.

Chéquense este twit – ¿cómo estaba el humor de la mujer que lo escribió?



Definitivamente andaba de malas.

Entender esto es un proceso sumamente sencillo para nosotros como seres humanos – incluso sin el contexto del twit, sin ver las expresiones faciales de la mujer o escuchar su entonación, por simple acto de leerlo ya sabemos cuál es su sentir.

Podríamos decir que estamos 99% seguros que estaba enojada – el otro 1% tal vez es que es un sarcasmo muy mal logrado.

Utilizando redes neuronales, podríamos pedirle a la computadora que haga lo mismo – leer un texto y que nos platique sobre los sentimientos de la persona que lo escribió. En este caso haremos un ejemplo extraordinariamente básico – reseñas de IMDB de películas (en inglés) y que nos diga si la reseña es positiva o negativa para la película.

La razón por la cual estamos haciendo este ejemplo tan mendigo inane (y aburrido en mi opinión) es porque… ustedes adivinaron … keras ya trae integrado un dataset de 50,000 reseñas de películas junto con un objetivo binario que nos dice si es negativa (0) o positiva (1).

Al igual que MNIST, el dataset de reseñas de IMDB es popular por buenas razones – lo puedes procesar en tu laptop de hace 7 años, pero te enseña buenas bases para el procesamiento de lenguaje natural.

### **Análisis de Sentimiento - Código**

Vamos cargando el dataset desde keras. Comenzando con la importación de las bibliotecas necesarias.

import tensorflow\_datasets as tfds

Ahora separaremos los datos para entrenamiento, validación y pruebas, la última línea hará que los datos los traiga tal cual como están etiquetados:

raw\_train\_set, raw\_valid\_set, raw\_test\_set = tfds.load(

    name = "imdb\_reviews",

    split =["train[:90%]", "train[90%:]", "test"],

    as\_supervised = True

)

Armaremos el dataset de train,. Vamos a barajear con buffer de 5,000 y lo vamos a batchear en 32 elementos con un prefetch para que siempre este a un paso adelante.

train\_set = raw\_train\_set.shuffle(5000, seed = 42).batch(32).prefetch(1)

Pasaremos a validación y prueba:

valid\_set = raw\_valid\_set.batch(32).prefetch(1)

test\_set = raw\_test\_set.batch(32).prefetch(1)

Vamos viendo unos reviews de muestra, para darnos una idea de lo que contienen. El raw\_train\_set hace que cada review venga individual, mientras que el normal viene por batch. Veamos que cada review viene con su respectiva etiqueta.

for review, label in raw\_train\_set.take(4):

print(review.numpy().decode("utf-8"))

print("label", label.numpy())

Texto

Descripción generada automáticamente

Ahora a entrenar el modelo, comenzaremos definiendo el tamaño del vocabulario. Armar nuestra capa de vectorización de texto utilizando keras.

vocab\_size = 1000

text\_vec\_layer = tf.keras.layers.TextVectorization(max\_tokens = vocab\_size)

Realizar la adaptación, esta solo tomara las reseñas y no las etiquetas:

text\_vec\_layer.adapt(train\_set.map(lambda reviews, labels: reviews))

La capa de TextVectorization no se le especificará que se realizará a nivel carácter, entonces se supondrá para su procesamiento que se deberá hacer a nivel palabra. Cada palabra tendrá u número. Que hayamos definido el tamaño de vocabulario como 1000 palabras le dirá al modelo que solo reconozca esas 1000 que son más usadas. Con el análisis de estas debería ser suficiente para saber si la reseña es buena o mala.

Armaremos el modelo. Haciendo que la review que introduzca tenga que pasar por **text\_vec\_layer**. Se usa función “sigmoid” debido a que nuestra tarea de clasificación es de dos elementos, si es buena o mala.

embed\_size = 128

model = tf.keras.Sequential([

    text\_vec\_layer,

    tf.keras.layers.Embedding(vocab\_size, 128),

    tf.keras.layers.GRU(128),

    tf.keras.layers.Dense(1, activation = "sigmoid")

])

Compilamos con CrossEntropy y ejecutamos a dos épocas.

model.compile(loss="binary\_crossentropy", optimizer="nadam",

              metrics=["accuracy"])

history = model.fit(train\_set, validation\_data=valid\_set, epochs=2)

Ahora evaluaremos el modelo injustamente porque solo le dimos dos épocas. Con un resultado de 0.6942, no hay sub ni sobre ajuste.

model.evaluate(test\_set)



Ahora una predicción, que traerá 32 resultados:

model.predict(test\_set.take(1))

Imagen que contiene Texto

Descripción generada automáticamente

Como podemos ver el modelo no hizo un buen trabajo por la cantidad de valores vacíos, está tomando las probabilidades prácticamente al azar.

Si imprimimos las etiquetas de cada review confirmamos lo mencionado en donde para la misma probabilidad en veces lo clasifica como bueno y en veces como malo.

for review,label in test\_set.take(1):

    print(label)



Ahora pasemos a masking para intentar mejorar el desempeño.

### **Masking**

Ahorita, el modelo va a tener que gastar ciclos valiosos de backpropagation aprendiendo que los tokens de relleno tienen que ser ignorados. Lo cual es un desperdicio, ya sabemos de entrada que debe de ignorarlos. Si tan solo hubiera una forma de pasarle esta regla para no quemar nuestro precioso GPU aprendiendo sandeces, nuestra vida sería más feliz.

Para eso usamos el Masking – en este caso mask\_zero = True le dice a la capa de Embedding que todos los tokens de relleno (que valen 0) serán ignorados por todas las capas rio abajo.

La manera en que esta función de Masking función es que la capa de embedding crea un tensor de mascar igual a K.not\_equal(inputs, 0): un tensor booleano con la misma figura que las entradas, y es igual a falso en cualquier lado los IDs sean 0s, verdadero de otra manera.

Este tensor de máscara se propaga al resto de modelo siempre y cuando se mantenga la dimensión de tiempo.

Cada capa va a manejar la máscara de manera independiente, pero en general simplemente ignoran los pasos enmascarados (ósea donde la máscara es falsa).

Por ejemplo, cuando una capa recurrente encuentra un paso enmascarado, simplemente copia la salida del paso previo. Si la mascar propaga hasta el final, entonces se va a aplicar a las perdidas también, así que los pasos enmascarados no contribuyen a la pérdida.

Todas las capas que reciben la máscara deben de soportar el masking – o vas a tener un error de excepción.

La capa Densa **Time Distributed**, todas las recurrentes, y varias otras soportan el masking.

En caso de que las capas que quieras usar no soporten el masking, pues que mala suerte, vas a tener que aventarte una programación de una versión modificada de tu capa que quieras usar para que soporte masking.

### **Masking – Código**

El masking es prácticamente lo mismo que ya vimos, pero esta vez cambiaran dos parámetros. Comenzamos de nuevo declarando el modelo, pero en el embedding ahora le daremos como parámetro mask\_zero = True, y con eso ignora los elementos enmascarados. Así quedaría:

embed\_size=128

model = tf.keras.Sequential([

    text\_vec\_layer,

    tf.keras.layers.Embedding(vocab\_size, embed\_size, mask\_zero = True),

    tf.keras.layers.GRU(128),

    tf.keras.layers.Dense(1, activation="sigmoid")

])

Se compila:

model.compile(loss="binary\_crossentropy", optimizer="nadam",

metrics=["accuracy"])

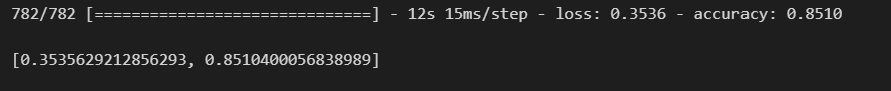
Lo entrenamos, esta vez a cinco épocas:

history = model.fit(train\_set, validation\_data=valid\_set, epochs=5)

Texto

Descripción generada automáticamente

En los resultados del entrenamiento podemos comparar la pérdida de validación entre los modelos, paso de 0.65 a 0.33.



A predecir los resultados sobre el primer set de datos:

model.predict(test\_set.take(1))

Texto

Descripción generada automáticamente

Ya se puede apreciar una clara diferencia en los que el modelo se decanta entre las reseñas positivas y negativas, con valores altos para los positivos como los primero con 0.9 y valores bajos para los negativos como el tercer y el cuarto. Esto se puede confirmar comparando con las etiquetas.

for review,label in test\_set.take(1):

    print(label)



## **Una red de Encoder - Decoder para Traducción mediante Redes Neuronales**

Vamos echándole un ojo a un simple modelo básico de traducción que agarra oraciones en inglés y las traduce al francés.

¿Por qué? Porque el primer sistema exitoso de traducción de máquina, el que desencadeno los traductores tales y cono los conocemos, fue diseñado para traducir del inglés a Frances.

Fue inventado en los 90s por un ingeniero de IBM llamado Bob Mercer (hoy multi billonario que se dedicó a las inversiones) cuando se le ocurrió la idea novedosa de entrenar un algoritmo de machine learning con el corpus más grande en existencia que cuenta con traducción perfecta – las minutas escritas de las sesiones parlamentarias canadienses.

Como ustedes tal vez sepan (o no) Canadá es un país que tiene 2 idiomas oficiales – el inglés y el francés, así que cualquier documento legal publicado por el gobierno tiene que ser distribuido en ambos idiomas. Si un ministro canadiense da un discurso en inglés, los secretarios deben de anotar la minuta en inglés y francés. Lo mismo aplica para algún ministro quebequense hablando en francés – el documento final sale en ambos idiomas.

Bob Mercer decidió entrenar a su algoritmo con estos documentos que ya estaban perfectamente traducidos y boom! Éxito total. Los ingenieros de antes intentaban entrenar a los programas de traducción enseñándoles la sintaxis, gramática y vocabulario de sus respectivos idiomas. Su tasa de éxito era lamentable.

Dado que los contribuyentes canadienses ya le regalaron a la comunidad de Data Science un Dataset para entrenar cualquier futuro algoritmo de traducción, es costumbre que muchos papers de traducción de máquina aprovechen este gran recurso y hagan ejemplos de inglés a Frances.

Nosotros no romperemos con esa gran tradición.

Vamos a estar haciendo una copia del modelo de Ilya Sutskever et al del 2014, en el que plantean un traductor de modelo encoder-decoder de inglés a francés. Ilya Sutskever también fue coinventor de AlexNet (recuerdan esta arquitectura de red convolucional del capítulo antepasado), y es el científico en jefe de Open AI.

Ah, y tiene solo 36 años en el 2022 y ya es líder en su campo. ¿Tú que has hecho de tu vida? (si te hace sentir mejor, ya está casi completamente calvo).

Aunque les voy a explicar a detalle cómo va a funcionar nuestro modelo de Encoder-Decoder, también les estoy dejando una copia en PDF del paper original de donde sale este modelo.

Muchachos, léanlo, en serio. Es lo que van a estar haciendo en sus chambas de Data Scientist de verdad. Y aparte, si no pueden entender un PDF que habla precisamente de estos temas, ¿para qué les sirve estar tomando el curso?

Hay vocabulario que no van a entender (Puntaje BLEU, dataset WMT-14, etc). Busquen en Google. En serio.

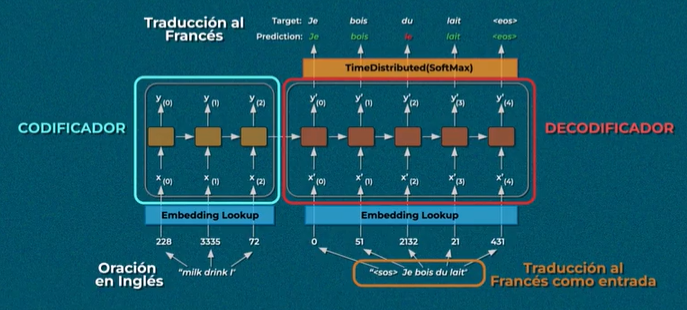
En la vida real no hay videos animados que te explican el tema sin letras griegas. [Link](https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf).

También, les voy avisando que nuestro entrenamiento de este modelo va a ser una mentada de madre de proporciones épicas para su intelecto. Solo para que se preparen para pedir el rembolso del curso (bromeo, a estas alturas del curso ya se les fue la oportunidad).

### **El codificador**

Para no hacerles el cuento largo, las oraciones en ingles se alimentan al codificador, y el decodificador saca las traducciones en Frances. Nota que las traducciones en Frances también se usan como entradas para el decodificador, pero movidas por un solo paso.

En otras palabras, el decodificador recibe como entrada la palabra que debió haber escupido en el paso previo (independientemente de lo que hubiera sacado). Para la primerísima palabra, solo recibe el token de inicio de secuencia.



El decodificador se espera que termine la oración con un token de fin de Oración.

Hay que tomar en cuenta que todas las oraciones en ingles están volteadas antes de ser alimentadas al codificador. “I drink milk” (yo tomo leche) entra como “milk drink I”. Si estamos hablando como Yoda de Star Wars es para asegurarnos que el inicio de la oración en ingles será alimentado al final al codificador, lo cual es útil porque generalmente es la primera cosa que el codificador debe traducir.

Cada palabra entra representada por su ID, y luego cuna capa de embedding regresa el embedding de la palabra en cuestión. Estos embeddings son los que en efecto son alimentados al codificador y el decodificador.

A cada paso, el decodificador saca un puntaje para cada palabra en el vocabulario de salida (Frances por ejemplo) y luego la capa de softmax cambia estos puntajes en probabilidades. Por ejemplo, en el primer paso la palabra “Je” puede tener una probabilidad del 20%, Tu puede tener una probabilidad del 1% y así.

La apalabra con la probabilidad más alta es sacada, esto es mucho como una tarea regular de clasificación, así que puede tomar el modelo usando perdida tipo Sparse categorical corssentropy – igual que como le hicimos con el modelo Char-RNN.

Toma en cuenta que cuando ya es momento de trabajar en el set de prueba (o en producción), no va a tener la sentencia de objetivo que alimentarle al decodificador. En vez de eso, simplemente vas a pasarle la palabra que saco en el último paso.

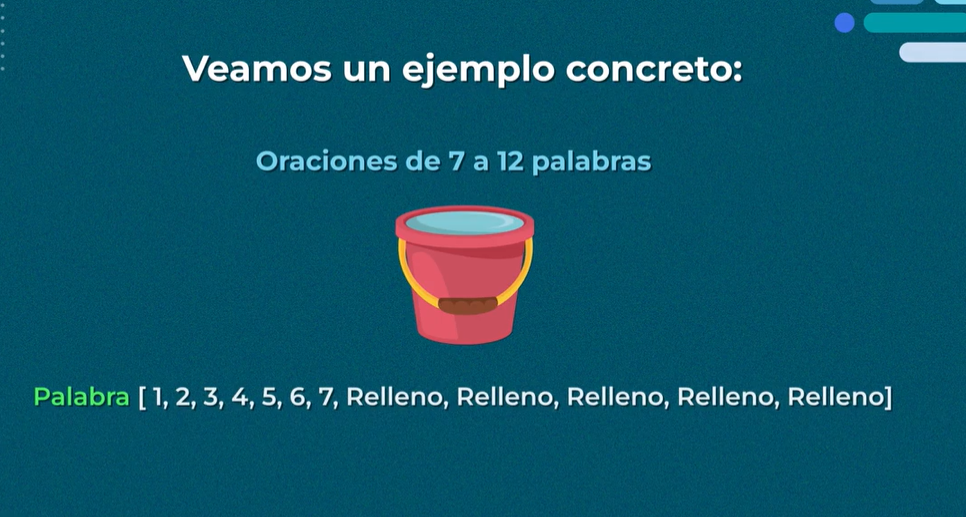
Listo, ya sabemos cómo funciona el sistema en general.

Dicho eso, el sistema requiere un poco más de detalles que necesitamos cubrir.

En primera, hemos estado suponiendo que todas las secuencias de entrada tienen un largo constante. Pero claro que los largos de oración varían. Ya que los tensores regulares tienen figuras fijas, pueden solo contener sentencias del mismo largo. Puedes usar masking para manejar esto, como lo discutimos antes. Dicho eso, si las sentencias tienen largos muy diferentes, no puedes simplemente recortarlas, ¡queremos traducciones completas!

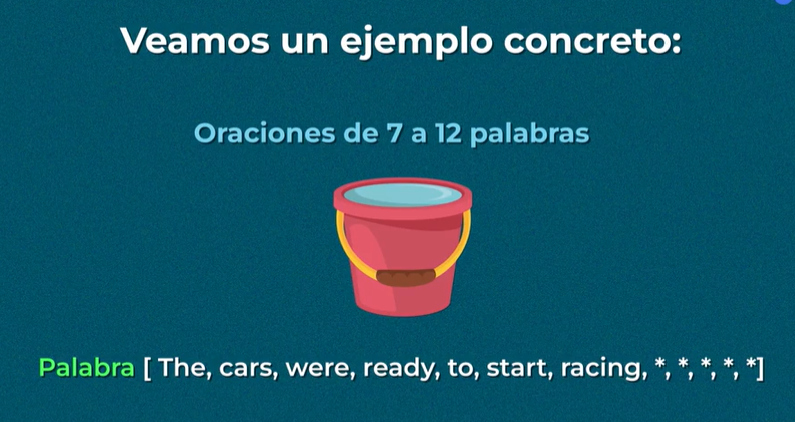
En vez de eso, vamos a agrupar las oraciones en cubetas de tamaños similares, usando rellenos para las secuencias más cortas y asegurando que todas las oraciones en una cubeta tienen tamaños similares.

Como ejemplo concreto – imaginemos que armamos una cubeta con oraciones de 7-12 palabras. Ya que queremos que todas las oraciones sean del mismo tamaño, simplemente vamos a usar rellenos para que las oraciones de 7 palabras acaben en 12 palabras. Le sumarias 5 rellenos.



Por ejemplo “The cars were ready to start racing” se convierte en “The cars were ready to start racing <relleno> <relleno> <relleno> <relleno> <relleno>”

Listo, si cuentan las palabras de la oración y los rellenos acabamos con 12 en total.



Otro tema que tomar en cuenta – queremos ignorar cualquier salida que venga después del token de fin de Secuencia, así que estos tokens no van a contribuir a la perdida (deben de usar mascara). Por ejemplo, si el modelo saca “les voitures étaient prêtes à commencer la course <FdS> Oui oui Baguette Paris Ohlalala Napoleon” Entonces todo lo que va del token en adelante se ignora.

Por último, hay que tomar en cuenta que nuestro vocabulario posible de salida es muy grande. Es todo el idioma francés (135,000 palabras según la edición actual de Dictionnaire de Francais). Así que estaríamos esperando que el decodificador escupa vectores de 135,000 dimensiones – eso significa que computar la función softmax sobre esta clase de vectores seria ridículamente intenso.

La manera más actual (en el 2022) de solucionar esto es ver solo la salida de *logits* del modelo para la palabra correcta, y para una muestra aleatoria de palabras incorrectas, y luego computar una aproximación de la perdida basándose en esta muestra de *logits*.

Este método se llama **Sampled Softmax** o softmax muestreado (en español).

Listo, es momento de armar nuestro modelo Encoder-Decoder – vamos a estar usando varias herramientas sacadas de **TensorFlow Addons**.

### **El Codificador – Código**

Comenzaremos preparando nuestro código con la importación de las bibliotecas necesarias, en este caso importaremos IO, y continuaremos con la lectura de nuestro set de datos para la traducción de inglés a español.

import io

read\_file= io.open("spa.txt", "r", encoding = "utf8")

text = read\_file.read()

Para visualizar alguno de los datos que tiene el archivo:

text[:100]



Continuaremos con la importación de Numpy y quitaremos los caracteres superflues en español, es decir, los que no aparecen en el inglés. Armaremos las parejas por líneas del set de datos y las barajearemos. Separaremos en oraciones en y oraciones es.

import numpy as np

text = text.replace("¡", "").replace("¿","")

pairs = [line.split("\t") for line in text.splitlines()]

np.random.seed(42)

np.random.shuffle(pairs)

sentences\_en, sentences\_es = zip(\*pairs)

Podemos imprimir tres oraciones del set de datos, para ver cómo están colocadas:

for i in range(3):

    print(sentences\_en[i], "=>", sentences\_es[i])

Texto

Descripción generada automáticamente

Definiremos el tamaño del vocabulario, queremos limitarlo y el tamaño máximo de caracteres en una oración:

vocab\_size = 1000

max\_length = 50

Ahora convertiremos las palabras de ingles a los índices numéricos. Trabajaremos todo en pares debido a los dos idiomas:

text\_vec\_layer\_en=tf.keras.layers.TextVectorization(vocab\_size, output\_sequence\_length=max\_length)

Ahora para las palabras en español:

text\_vec\_layer\_es=tf.keras.layers.TextVectorization(vocab\_size, output\_sequence\_length = max\_length)

Usaremos **.adapt** para computar un vocabulario de strings desde los tokens en tu vocabulario vectorizado. Este proceso puede durar varios minutos, para inglés:

text\_vec\_layer\_en.adapt(sentences\_en)

Ahora para el español:

text\_vec\_layer\_es.adapt(([f"startofseq {s} endofseq" for s in sentences\_es]))

Con esto adaptaremos todo el set de datos del español y también los tokens de “start” y “end” de una oración. Esto es importante debido a que el descodificador estará tomando la oración objetivo y los tokens son necesarios para el algoritmo.

Podemos imprimir las diez palabras más comunes del set de inglés:

text\_vec\_layer\_en.get\_vocabulary()[:10]



Las de español:

text\_vec\_layer\_es.get\_vocabulary()[:10]



Ahora definiremos nuestro set de entrenamiento, validación y prueba como en todos los demás modelos que hemos estado haciendo.

X\_train = tf.constant(sentences\_en[:100\_000])

X\_valid = tf.constant(sentences\_en[100\_000:])

X\_train\_dec = tf.constant([f"startofseq {s}" for s in sentences\_es[:100\_000]])

X\_valid\_dec = tf.constant([f"startofseq {s}" for s in sentences\_es[100\_000:]])

Y\_train = text\_vec\_layer\_es([f"{s} endofseq" for s in sentences\_es[:100\_000]])

Y\_valid = text\_vec\_layer\_es([f"{s} endofseq" for s in sentences\_es[100\_000:]])

Define las capas de entradas del encoder y del decoder.

tf.random.set\_seed(42)

encoder\_inputs = tf.keras.layers.Input(shape=[], dtype = tf.string)

decoder\_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)

Ahora asígnales tus textos vectorizados a esas capas nuevas.

embed\_size = 128

encoder\_input\_ids = text\_vec\_layer\_en(encoder\_inputs)

decoder\_input\_ids = text\_vec\_layer\_es(decoder\_inputs)

Prepara una capa de embedding cada uno (decoder y encoder).

encoder\_embedding\_layer = tf.keras.layers.Embedding(vocab\_size, embed\_size, mask\_zero = True)

decoder\_embedding\_layer = tf.keras.layers.Embedding(vocab\_size, embed\_size, mask\_zero=True)

Y ahora mete tus input\_ids a la capa de embedding.

encoder\_embeddings = encoder\_embedding\_layer(encoder\_input\_ids)

decoder\_embeddings = decoder\_embedding\_layer(decoder\_input\_ids)

Define el encoder como 1 sola capa LSTM de 512 neuronas, pasa tus embeddings por esa capa y escupe los outputs y el state.

encoder = tf.keras.layers.LSTM(512, return\_state = True)

encoder\_outputs, \*encoder\_state = encoder(encoder\_embeddings)

Repite el proceso correspondiente para el decoder.

decoder = tf.keras.layers.LSTM(512, return\_sequences = True)

decoder\_outputs = decoder(decoder\_embeddings, initial\_state = encoder\_state)

La capa de salida será una capa Densa del mismo tamaño que el vocabulario con activacion softmax, nos escupira la probabilidad Y de alguna palabra tomando los decoder outputs como entrada.

output\_layer = tf.keras.layers.Dense(vocab\_size, activation="softmax")

Y\_proba = output\_layer(decoder\_outputs)

Arma tu modelo.

model = tf.keras.Model(inputs = [encoder\_inputs, decoder\_inputs], outputs = [Y\_proba])

Compilamos.

model.compile(loss="sparse\_categorical\_crossentropy", optimizer="nadam",

              metrics=["accuracy"])

Arma un checkpoint para guardar el modelo.

tmodel\_ckpt = tf.keras.callbacks.ModelCheckpoint(

    "traductor\_modelo", monitor="val\_accuracy", save\_best\_only=True)

Ejecuta a 10 épocas, nota que tienes que meter 2 sets de entrenamiento y validación para X (enc y dec) - se va a tardar como ¡20 minutos por época sin GPU! (1 min con gpu). Esto solo entrenara el modelo.

history=model.fit((X\_train, X\_train\_dec), Y\_train, epochs = 10, validation\_data=((X\_valid, X\_valid\_dec), Y\_valid), callbacks = [tmodel\_ckpt])

Vamos ahora a armar el traductor

def translate(sentence\_en):

    translation = ""

    for word\_idx in range(max\_length):

        X = np.array([sentence\_en])

        X\_dec = np.array(["startofseq" + translation])

        y\_proba = model.predict((X, X\_dec))[0,word\_idx]

        predicted\_word\_id=np.argmax(y\_proba)

        predicted\_word =text\_vec\_layer\_es.get\_vocabulary()[predicted\_word\_id]

        if predicted\_word == "endofseq":

            break

        translation += " " + predicted\_word

    return translation.strip()

Prueba del traductor con:

translate("I like soccer")



Podemos ver que esta traducción se desempeña muy bien.

Prueba con una oración larga como “I like soccer and going to the beach”

translate("I like soccer and also going to the beach")

### **RNNs Bidireccionales**

A cada paso de tiempo, una capa recurrente regular solo se fija en las entradas del pasado y el presente antes de genera la salida. En otras palabras, es “Causal”, ósea no puede ver al futuro.

Este tipo de RNN tiene todo el sentido del mundo cuando estás prediciendo series de tiempo, pero para muchas tareas de NLP, como la traducción de Máquina, generalmente es preferible ver las siguientes palabras antes de codificarlas. Por ejemplo, considera el contexto de las frases “La madre de todas las batallas”, “Le dieron en su madre al pobre”, “La Madre del licenciado se identificó con su pasaporte” y “Madre santísima apiádate de nosotros”.

Para poder codificar la palabra Madre apropiadamente en cada una de estas frases, necesitas conocer el contexto completo de la oración que rodea a la palabra Madre.

Para poder implementar esto, hay que aplicarles 2 capas recurrentes a las mismas entradas, una leyendo las palabras de izquierda a derecha y la otra leyéndolas de derecha a izquierda. Luego simplemente combina las salidas en cada paso de tiempo, generalmente por concatenación. Esto se llama capa recurrente bidireccional.

Una captura de pantalla de un videojuego

Descripción generada automáticamente con confianza baja

### **RNN Bidireccional - Código**

Hacer esto en keras es la cosa más sencilla del mundo – ya trae su propia envoltura. Ahora armaremos un modelo secuencial con una capa GRU y una capa GRU bidireccional.

encoder = tf.keras.layers.Bidirectional(

    tf.keras.layers.LSTM(256, return\_state=True))

### **Beam Search**

Vamos a suponer que entrenas un Codificador – Decodificador, y lo usas para traducir la oración en Frances “Comment vas tu?” al español. Se supone que te va a escupir, si supiera lo que hace, ¿“Como estás?”, pero desafortunadamente te devuelve “¿Como vas tu?”.

Si ves el set de entrenamiento vas a ver muchas oraciones como “Comment vas tu jouer?” que se traduce a “Como vas a jugar tu?”. Así que no estaba fuera de lugar que el modelo tradujera el “Comment vas” a “como vas” en español.

Desafortunadamente en este caso fue un error, y el modelo no pudo volver y arreglarlo, así que intento completar la oración como mejor pudo.

Ya que se obsesionó a intentar obtener la mejor palabra en cada paso, sin fijarse en el contexto completo, termino con una traducción subóptima de la oración final.

¿Hay maneras de permitirle al modelo echarse para atrás y arreglar el error que cometió antes?

La solución más común es **Beam Search** (Aunque la verdad no hay una traducción directa en español, valdría la pena pensar en Beam en este caso como Viga o Polín).

Beam Search mantiene una lista corta de una cantidad de las oraciones más prometedoras. Aunque esta cantidad la defines tu (ósea las k oraciones más prometedoras), podríamos imaginarnos que estamos usando el top 3 de oraciones. En cada paso del decodificador intenta extenderlas por una palabra, manteniendo solo las oraciones K más probables. El parámetro K se llama ancho de Beam (viga).

Por ejemplo – vamos a usar el modelo para traducir “Comment vas tu?” usando beam search con un ancho de 3. Al primer paso de decodificador el modelo va a sacar una probabilidad estimada para cada palabra posible. Supongamos que las top 3 de palabras son “como” (75%), “que” (3%) y “Tu” (1%). Esa es nuestra lista hasta ahorita.

Ahora, crearemos 3 copias de nuestro modelo y las usaremos para encontrar la siguiente palabra para cada oración. Cada modelo va a sacar una probabilidad estimada por palabra en el vocabulario. El primer modelo va a intentar encontrar la siguiente palabra en la oración “como” y tal vez sacar una probabilidad de 36% para la palabra “vas”, 32% para la palabra “son” y 16% para hacer, y así.

Nota que son probabilidades condicionales, dado que la oración empieza con “Como”, tal vez saca una probabilidad condicional del 50% para la palabra “son” y así.

Suponiendo que el vocabulario tiene 10,000 palabras, cada modelo va a sacar 10,000 probabilidades.

Luego, computamos las probabilidades de cada una de las 30,000 oraciones de dos palabras que estos modelos consideraron (3 x 10,000). Hacemos esto multiplicando la probabilidad condicional estimada de cada palabra por la probabilidad estimada de la oración que completa.

Por ejemplo, la probabilidad estimada de la oración era de 75%, mientras que la probabilidad condicional estimada de la palabra “vas” era de 36%, así que la probabilidad estimada de la oración “como vas” es de 75% x 36% = 27%.

Después de computar las probabilidades de todas las 30,000 oraciones de dos palabras, mantenemos solo las top 3. Además, todas empiezan con la palabra “Como”: “Como vas” (27%) “Como estas” 24% “Como haces” (12%) – ahorita la oración “como vas” está ganando, pero “como estas” no ha sido eliminada.

Y luego repetimos el mismo proceso: usamos tres modelos para predecir la siguiente palabra en cada una de estas tres oraciones, y luego computamos las probabilidades de todas las 30,000 oraciones de 3 palabras. Tal vez el top tres ahora son: “Como estas tu?” (10%) “Como haces tu” (8%), “Como harás tu” (2%). Al siguiente paso obtendremos “Como te va a ti”, “Como estas tu EOS”, y “Como te harás tu esto”, nota que “Como vas” ya fue eliminada y tenemos 3 traducciones perfectamente razonables.

## **Mecanismos de Atención**

Considera el camino de la palabra “Leche” (Milk) a su traducción “lait” en un esquema de codificador Decodificador normal.

Se convierte en número, pasa al embedding, entra en la primera celda recurrente del RNN y luego tiene que esperar a que se codifique el resto de la oración. Posteriormente – sale al otro lado y se espera a que se decodifique la oración traducida y al final, en la penúltima iteración del descodificador, acabamos con la palabra “lait”.

Es bastante largo el camino, y eso que la oración es sumamente sencilla. “I drink Milk”. El principal problema es que debemos de estar cargando una representación de esta palabra a través de muchísimos pasos antes de darle uso. ¿No podemos hacer este camino más corto?

Esta es la idea fundamental detrás de la atención. Propuesta en el 2014 por Dzmitry Bahdanau et al.

Les estoy dejando una liga al [paper](https://arxiv.org/pdf/1409.0473.pdf) original porque, adivinaron, su tarea va a consistir en darle una leída.

No está sencillo, toda la estructura de codificador-descodificador que acabamos de ver se la echa en términos matemáticos.

Y bueno, una vez que lo lean, toman sus notas, y lo meditan, vuelven acá con nosotros para seguir con el tema de la atención.

Dzmitry Bahdanau introdujo una técnica que permitió al decodificador enfocarse en las palabras apropiadas (codificadas por el codificador) en cada paso de tiempo. Por - ejemplo, en el lapso donde el decodificador necesita sacar la palabra “lait”, va a enfocarse en el apalabra “leche”. Esto significa que el camino desde una palabra de entrada a su traducción es ahora mucho más corta, así que las limitaciones de memoria de corto plazo ahora tienen mucho menos impacto.

Estos mecanismos de atención han sido una revolución para el procesamiento de lenguaje natural: permiten una mejora significativa en el estado del arte, especialmente para oraciones largas (más de 30 palabras).

¿Cómo es la arquitectura el modelo?

Primero vamos a comenzar con la estructura tradicional – codificador y decodificador. El codificador va a recibir la señal, pero en vez de pasarle el estado oculto final al decodificador, ahora mandamos todas las salidas al decodificador.

En cada paso de tiempo, la celda de memoria computa una suma ponderada de todas estas salidas de descodificador: esto determina cuales palabras se va a enfocar en este paso.

El peso Alpha(t,i) es el peso de la salida i-ésima del codificador en el paso T de tiempo del decodificador.

Por ejemplo, si el peso α(3,2) es mucho más grande que los pesos α(3,0) y α(3,1), entonces el decodificador va a poner mucha más atención a la palabra #2 (leche) que a las otras 2 palabras, al menos en este paso de tiempo. El resto del decodificador funciona igual que antes, en cada paso de tiempo la celda de memoria recibe las entradas que acabamos de discutir, más el estado oculto del estado oculto previo, y finalmente, recibe la palabra objetivo del estado previo.

¿Aquí la duda es, de donde salen estos pesos Alpha?

Es muy sencillo, son generados por un tipo de pequeña red neuronal llamada “Modelo de Alineación” (o capa de atención) que se entrena junto con el resto del modelo Codificador Decodificador. Este modelo de alineación comienza con una capa densa distribuida por tiempo de una sola neurona, que recibe como entrada todas las salidas del codificador, concatenadas con el estado oculto previo del decodificador.

Finalmente, todos los puntajes pasan por una capa de softmax para obtener un peso final para cada salida del codificador.

Todos los pesos para un dado paso de tiempo del decodificador suman a uno (ya que la capa de softmax no está distribuida por tiempo). Este mecanismo de atención en particular se llama Atención Bahdanau. También le llaman atención concatenativa por que concatena la salida del codificador con el estado previo del codificador.

Otro mecanismo de atención fue propuesto por Minh Than Luong et al en el 2015. [Link](https://arxiv.org/pdf/1508.04025.pdf).

Les dejo la liga también porque también es mi intención que lean el paper (ándenle, ya le deberían de poder entender).

Tristemente, la atención Bahdanau a la que tanta atención le pusieron no agarro tanta tracción como la atención Luong – de hecho, los mecanismos que surgen para potencializar la atención y hacerla más efectiva se basan en el método de Luong.

¿Como funciona el método Luong?

Fácil, ya que la meta del mecanismo de atención es medir la similitud entre una de las salidas del codificador y el estado oculto previo del decodificador, los autores propusieron simplemente computar el producto punto de estos 2 vectores, y resulto ser que esta es una medida bastante buena de similitud. También, es muy fácil para las computadoras modernas hacer esta clase de operación.

Para que esto se pueda dar, ambos vectores deben tener la misma dimensionalidad.

Aparte de llamarse Atención Luong, se puede llamar Atención Multiplicativa.

El producto punto nos da un puntaje, y todos los puntajes (en algún momento dado del descodificador, pasan por una capa de softmax para darme los pesos finales, igual que en la atención Bahdanau.

Otra simplificación que sale de la atención Luong es usar el estado oculto del descodificador en el momento actual antes que, en el momento previo, y luego usar la salida del mecanismo de atención directamente para computar las predicciones del decodificador.

También propusieron una variante del mecanismo de producto punto donde el codificador primero pasa a través de una transformación lineal antes de que se calculen los productos punto.

Para echarle sal a la herida de Bahdanau, compararon sus métodos multiplicativos y demostraron, conclusivamente, que la atención Luong es superior.

Así que Bahdanau invento el concepto de atención, sí, pero nadie usa su método hoy por hoy – todo es producto punto.

### **Atención – Código**

Envuelve tu encoder en una capa bidireccional, como hace rato.

encoder = tf.keras.layers.Bidirectional(

tf.keras.layers.LSTM(256, return\_sequences = True, return\_state = True))

Vamos a volver a armar esta parte del modelo, casi igualita a la anterior, para que cache nuestra nueva onda bidireccional.

encoder\_outputs, \*encoder\_state = encoder(encoder\_embeddings)

encoder\_state = [tf.concat(encoder\_state[::2], axis=-1),tf.concat(encoder\_state[1::2], axis=-1)]

decoder = tf.keras.layers.LSTM(512, return\_sequences=True)

ecoder\_outputs = decoder(decoder\_embeddings, initial\_state=encoder\_state)

Armamos las capas de atención en keras. Esta ya está pre hecha gracias a Dios.

attention\_layer = tf.keras.layers.Attention()

attention\_outputs = attention\_layer([decoder\_outputs, encoder\_outputs])

output\_layer = tf.keras.layers.Dense(vocab\_size, activation="softmax")

Y\_proba = output\_layer(attention\_outputs)

Y terminamos el modelo

model = tf.keras.Model(inputs=[encoder\_inputs, decoder\_inputs],

                       outputs=[Y\_proba])

Compilamos:

model.compile(loss="sparse\_categorical\_crossentropy", optimizer="nadam",

              metrics=["accuracy"])

Corremos el modelo, alrededor de 3 min por ejecución.

model.fit((X\_train, X\_train\_dec), Y\_train, epochs=10,

          validation\_data=((X\_valid, X\_valid\_dec), Y\_valid))



Tuvimos una pérdida de validación de 0.19 vs el anterior de 0.20

Si volvemos a probar con la traducción de: “I like soccer and also going to the beach”



### **Atención es todo lo que necesitas – Transformers**

Al final podríamos decir que la intención de todo este capítulo es llevarnos a conocer a los Transformers. No, tristemente no los Autobots y los Decepticons (y se aguantan con mis chistes malos). Los Transformers son una arquitectura revolucionaria en el ámbito del procesamiento de lenguaje natural mediante redes neuronales.

No sé si han estado viviendo bajo una piedra, pero existe un algoritmo llamado GPT-3 que puede escribir ensayos y contestar preguntas en lenguaje natural con una calidad muy superior a tu alumno promedio de universidad. Le puedes pedir que te explique por qué un chiste en particular causa gracia o porque cierto refrán hace sentido… y GPT-3 te va a contestar mejor que muchos humanos.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Muchachones, esto es una revolución – llegamos al punto en el que un alumno puede pasarle su pregunta de ensayo a la computadora, obtener un escrito original de regreso, y recibir un 8 o 9 de calificación por parte del profesor.

Aparte de GPT-3 existen otras arquitecturas que le hacen competencia, como BERT, que también tienen capacidades increíbles en el ámbito de procesamiento de lenguaje natural. Y nosotros en este curso estamos a punto de terminar de entender cómo funcionan estas redes que son la punta de lanza tecnológica en pleno 2022.

El bloque básico que hace todo esto posible es el **Transformer**.

Diagrama

Descripción generada automáticamente

Presentado en el 2017 por un equipo de científicos de Google en un [paper](https://arxiv.org/pdf/1706.03762.pdf) llamado “Attention is all you need – Transformers” en el que logran armar esta joyita de arquitectura sin absolutamente ninguna capa recurrente o convolucional. Y como extra – esta arquitectura era mucho más rápida de entrenar y fácil de paralelizar, así que la lograron entrenar a una fracción del tiempo y costo que los modelos previos.

¿Y para que llevamos 2 capítulos viendo redes recurrentes si los Transformers las volvieron obsoletas?!?!?!? Te estarás preguntando ahorita, todo enojado.

Ahorita vamos a ver que los Transformers funcionan igual sobre la arquitectura de codificador decodificar que hemos estado viendo – de hecho, son un encoder-decoder común y corriente, solo que sin las redes neuronales recurrentes.

También, el elemento fundamental en un transformer es la atención, que es lo que acabamos de ver hace ratito. Si no pasábamos por RNNs, no podíamos llegar a esto.

Y, por último, las arquitecturas van cambiando – hoy es Transformers, tal vez el día de mañana alguien encuentra otra red mucho muy superior a base de RNNs, y luego alguien más sale con una mejora creada con puras convolucionales. Por eso es importante conocer los bloques básicos de las diferentes redes neuronales.

Ok, vamos comenzando – este es el diagrama original del Transformer como se nos presentó en el paper de Attention is all You need.

Si se fijan, se ve clara la estructura del codificador, de este lado, y el decodificador, de este otro.

Muchos de los elementos ya los conocen, capas densas, capas de normalización, conexiones de skip, sofmax y capas lineales.

Pero hay muchas otras que no conocen y son fundamentales para su funcionamiento.

Así que vamos comenzando por el principio – desde abajo ¿Qué come un transformer? ¿Como le alimentamos los datos?

### **Que comen los Transformers.**

En este video comenzaremos hablando las entradas que le pasamos al codificador del transformer – primero el procesamiento básico de la entrada, luego como se transforma en embeddings, y luego terminamos con el concepto importantísimo de embedding de posición, que hace que toda esta magia funcione.

Vamos a imaginar para este ejemplo que estamos haciendo un robotito que completa los diálogos más citables de las películas.

Le alimentamos miles de diálogos de películas y debe de aprender a recibir un dialogo a medias, de una película que no ha visto, y completarlo.

Ok, para mantener un tono ligero y amistoso, la película en este caso será “El Padrino 2” y la oración que queremos completar es la clásica de Hannibal Lecter “Keep your friends close.” (Mantén a tus amigos cerca)

Si saben de buen cine, sabrán que esta frase termina “And your enemies closer”. (Y a tus enemigos más cerca)

Asi que la entrada será “Keep your Friends close”, esta es la frase que le metemos al codificador.

Ahora, ustedes ya saben que las computadoras son totalmente incapaces de entender palabras en inglés y tenemos que tokenizarlas para convertirlas a números. Así que ya se la saben.

Tomamos todas nuestras palabras presentes en nuestro set de entrenamiento, en este caso todo el vocabulario de todas las películas que usamos para entrenar a nuestro completador, y generamos un diccionario donde a cada palabra le corresponde un número.

Podríamos usar el mismo método que aplicamos para IMDB, en el que rankeamos las palabras por popularidad, y le asignamos los números más bajos a las palabras más populares.

Siguiente, vamos a escoger las palabras/números que suceden en el texto de entrada “Keep your Friends close”. Vamos a denotar estas entradas como variable con la letra x. Y listo, esto es lo que vamos a meter al codificador del transformer.

Ahora si – ya acabamos con el preprocesamiento, es hora de meternos a cómo funciona el transformer.

Si se fijan ya completamos la parte de “inputs” en este diagrama, ósea lo que sigue es “Input embedding”.

Aunque se supone ya sabemos que es una capa de embedding, vamos repasando: La capa de embedding tiene un índice para cada palabra en nuestro vocabulario, y contra cada uno de esos índices se le asigna un vector.

Inicialmente, estos vectores se llenan de números totalmente aleatorios. Después, durante las fases de entrenamiento el modelo las actualiza con valores que los ayudan mejor con las tareas asignadas.

Ahora, yo he escogido un tamaño de embedding de 3 para que sea fácil graficarlos en el ejemplo y quepan en pantalla. Pero les aviso que en el transformer original el embedding es de tamaño 512.

Así que – ¿qué es un embedding? Bueno son simplemente representaciones vectoriales de una palabra dada. Cada dimensión del embedding intenta capturar algún elemento lingüístico sobre esa palabra y estos podrían ser cosas como si la palabra es un verbo, un sustantivo o algo más. Pero en realidad esto es racionalización humana, el modelo se va a inventar sus propios criterios que tal vez no entendamos.

Muchas veces es muy difícil figurarse que representa cada una de estas dimensiones.

Gráficamente los valores de estas dimensiones representan las coordenadas de la palabra en un plano hiperespacial – si 2 palabras comparten características lingüísticas similares, y aparecen en contextos similares, sus valores de embeddings se actualizan para acercarse más y más durante el entrenamiento.

Por ejemplo, considera las 2 palabras raqueta y pelota – al inicio sus embeddings están dados al azar, pero durante el tenrenamiento se van haciendo más y más similares ya que estas palabras siempre aparecen en contextos similares, vs cuando comparamos estas 2 contra una tercera palabra como ventilador. Ventilador va a aparecer en contextos muy diferentes durante el entrenamiento.

Y así, hacia el final del entrenamiento su representación gráfica se puede ver así.

Por ende, la capa de embedding selecciona el embedding que corresponde a la prueba de entrada y lo pasa hacia adelante. Vamos a denotar estos embeddings con la letra e.

Recapitulando, la capa de embedding toma los índices de entrada y los convierte en embeddings de palabra, y luego estos se pasan hacia la siguiente capa.

Vamos a seguir adelante y visitar el componente final que discutiremos en el video – los embeddings de posición.

¿Por qué necesitamos meter este elemento?

Imagina que tenemos esta celda de GRU, si agarra estos embeddings tomaría uno por uno de manera secuencial (primero el uno, luego el 2, luego el 3 y asi), por lo que su entrenamiento va a resultar ser lento.

Dicho eso, no todo es malo, ya que los GRUs toman sus embeddings de manera secuencial, ellos saben cuál palabra vino primero y cual palabra vino después y así. Los Transformers por su parte agarran todos los embeddings al mismo tiempo – esto es buenísimo para los tiempos de entrenamiento, pero pues, se pierde la información sobre el orden de las palabras.

Para los ancianos tomando el curso – es como cuando ibas a instalar un programa que venía en 100 diskettes de floppy sin etiquetar… y el wey que los traía se le caían al piso. ¿Ahora cuál era el primer disco, el segundo y el tercero y así? Imposible volver a saber.

Lo mismo pasa cuando el Transformer agarra los embeddings todos al mismo tiempo.

No necesito decirles que esto es un problemón severo – el lenguaje necesita de cierto orden para ser entendido. No es lo mismo huele a traste, que atrás te huele.

Entonces ¿cómo le podemos hacer para mantener el orden del transformer sin tener que meterles una red neuronal recurrente, como un GRU?

Lo primero que haremos será introducir un nuevo set de vectores que contienen la información de posición – y los llamaremos embeddings de posición. Comenzamos simplemente sumando los embeddings de palabra que ya teníamos con sus correspondientes embeddings de posicione y crear un nuevo embedding que está consciente del orden de las palabras.

¿Y qué le vamos a meter a los embeddings de posición nuevos que armemos?

Podríamos intentar literalmente meterle el número de posición de palabra, así que el primer embedding de posición está lleno de 0s, el segundo está lleno de 1 y luego 2s y así consecutivamente.

El problema es que meterle números completos así podría distorsionar la información del embedding de palabras original, que la verdad, es el que nos importa. Por ejemplo, si tenemos una frase de 40 palabras, estos últimos embeddings de posición van a tener valores de 38,39,40 – y eso solo nos generaría problemas. Para la modelo seria 1000 veces más importante la posición de la palabra que su contexto.

Ok, podrías estar pensando que entonces le metemos fracciones, así si nuestro texto es de 4 palabras, entonces nuestros embeddings de posición simplemente van a representar la posición de la palabra como una fracción del largo total, de esa manera el valor del embedding nuca será mayor a 1.

Entonces no tendríamos el problema anterior de tener embeddings de posición que opacan a nuestros embeddings de palabra. ¿Correcto?

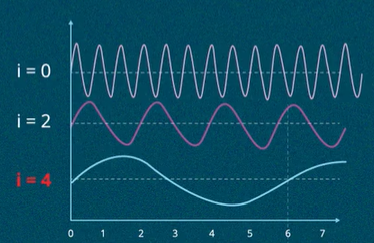
El problema es que este método tampoco funciona por razones diferentes. El hacer los embeddings una función del largo total significa que, si las oraciones que estas metiendo difieren en el largo, cosa que siempre hacen, entonces tendrían diferentes embeddings de posición para la misma posición.

Esto acabaría por confundir al modelo y pues obviamente no es lo que estamos buscando.

Idealmente los valores de embedding de posición en alguna posición dada deberían de quedarse igual sin importar el largo total del texto o cualquier otro factor. Así ¿qué estaríamos haciendo? Bueno los autores del paper original sugirieron, así sin que nadie les preguntara, utilizar funciones de onda para capturar información de posición – veamos cómo.

Primero, vamos a tomar el primer embedding de posición como ejemplo.

La variable de posición en esta fórmula será de 0. Siguiente, el tamaño del embedding de posición tiene que ser el mismo que el de los embeddings de palabra, así que vamos a ponerlo a 3 dimensiones. Lo vamos a representar por la letra de en nuestra formula. La letra i aquí representa el índice de cada uno de los embeddings de posición.



Ahora, si dibujamos una curva senoidal, vamos a obtener una curva que se suave, continua como la que estamos mostrando aquí. Así que vamos colocando varios de nuestros embeddings de posición ahora ya que el alto de la curva de seno depende de la posición en el eje x.

Podemos usar el alto de la curva como un reemplazo para nuestras posiciones ya que el alto de la curva solo varía entre un rango fijo y no depende del largo del texto de entrada.

Este método nos puede ayudar a superar la limitación previamente discutida.

Hay un problema, sin embargo; nota como el embedding de posición de las palabras en la posición 0 y 6 son exactamente las mismas incluso cuando idealmente deberían ser diferentes, ya que 0 y 6 son 2 posiciones diferentes.

Esto es donde entra la siguiente variable en la ecuación, la i. El sesgo rojo si vuelves a graficar la curva en diferentes valores de i, la idea es que obtienes una serie de curvas de diferentes frecuencias.

Ahora, este es el punto de tener curvas de diferentes frecuencias. SI 2 puntos están cerca en la curva, van a estar idénticos en frecuencias más altas también. Solo es en frecuencias mucho más altas que su coordenadas-y en la curva son diferentes.

Y puede que las puedas diferencia para los puntos más lejanos. Por otro lado, deberías de ser capaz de empezar a verlos caer en diferentes alturas de curva muy antes, así que ambas la posición así como la dimensión del embedding pueden darnos información del orden.

Vamos haciendo esto más concreto viendo un ejemplo, supongamos que nos gustaría leer los embeddings de posición para las palabras en los lugares 0 y 6. Ahora si lees el valor del embedding de posición en la posición 6 y la dimensión i es igual a 4, en las frecuencias más bajas lo vas a encontrar que va a ser igual al de p0.

Estos van a empezar a ser diferentes de manera significativa cuando vas a dimensiones que corresponden a frecuencias más altas, y así los vectores finales de cada posición pueden ser muy diferentes. Ahora los autores no usaron solo las curvas de seno, usaron una combinación de fórmula de seno y coseno.

Los autores originales del paper usan una combinación alternativa de seno y coseno para armar las curvas en posiciones. La curva de seno es para los embeddings de posición nones, y la curva de coseno es para los embeddings pares.

Así se ven las curvas de posición cuando son graficadas en escala completa, así que para obtener el valor de un embedding de posición en cierta dimensión simplemente puedes leerlo desde la gráfica.

Entonces ya que tenemos esto, podemos sumar nuestros embeddings a los embeddings de palabra y listo, ya podemos pasarle los datos al codificador.

En el siguiente video veremos como el modelo usa la **autoatención** para que funcione el transformer.

### **Atención Multi – Cabeza**

Decidimos construir un modelo para completar el dialogo automáticamente usando Transformers, y lo usaremos para completar la frase de El Padrino 2 “keep your friends close, and your enemies closer”.

Para hacer eso pasamos el texto de entrada a la capa de embedding, y luego le sumamos la información de la posición a nuestros embeddings de palabra. Los embedding resultantes estaban conscientes de la posición y fueron pasados a la siguiente capa, capa que estaremos explorando en este video.

Esta capa se llama en inglés **MULTI HEAD ATTENTION** (Atención Multi-Cabeza en español) y es el elemento más importante de un transformer. Este tipo de capa fue inventando específicamente para los Transformers, no existía anteriormente.

Vamos recordando un poquito sobre la Atención.

Si les paso la siguiente frase – “Ciudad Fundada por Hernán Cortez en 1519 para marcar el sitio de su desembarco final en México en honor al lugar donde murió Cristo”.

¿Me pueden decir de que ciudad estamos hablando? Ok, si saben un poco de historia de México si – si vienen de otros países pues ehmm…. éxito muchachos.

Para contestarme esa pregunta probablemente se tengan que fijar más en ciertas palabras que otras. Le vas a poner especial *atención* a las palabras Hernán Cortez, 1519, lugar donde murió Cristo.

Y sabrías que esta ciudad no es ninguna otra más que Veracruz – como era su nombre original La Villa Rica de la Vera Cruz.

Vera es Verdadera y Cruz pues… no necesita explicación.

Así que un mecanismo de atención le ayudaría a un modelo enfocarse en las palabras más importantes en una entrada dado.

Dicho eso, los Transformers, como les dije, no lograron lo que lograron con simple atención – el equipo de los Transformers decidió usar un concepto se llama en ingles **self-attention** o autoatención.

Para explicar esto de la autoatención es más fácil con un ejemplo.

Considera la siguiente oración:

El Cabo Márquez, recién ascendido, contempló el horizonte desde su nuevo puesto de vigilancia en Cabo San José; mientras el sol se ponía, se dedicó a recoger el cabo grueso que estaba flotando en la orilla del muelle.

Considera que la misma palabra “cabo” tiene 3 significados diferentes: La primera instancia de “Cabo” se refiere al rango militar que va a 1 arriba de soldado raso, la segunda instancia se refiere a un lugar geográfico tipo península. Y la última instancia de Cabo significa cuerda náutica para amarrar barcos.

Entonces, aquí la duda es como puede saber el modelo a que se refiere cada una de las incidencias de Cabo.

La manera en la que los humanos lo hacemos es juzgar el significado de la palabra poniéndole atención al contexto en la que aparece. Por ejemplo – Cabo Márquez, recién ascendido, tal vez estamos hablando de un militar. El hecho de que “contempla el horizonte” nos dice que definitivamente no es un lugar geográfico. Las penínsulas no pueden contemplar.

Puesto de Vigilancia en Cabo San José – estamos dándole indicios de que esto es un lugar – porque puesto de vigilancia es un lugar, y está en Cabo San José. Una Península si puede tener puestos de vigilancia.

Por último – Cabo grueso que estaba flotando en la orilla del muelle es significativamente más difícil. ¿Puede un militar grueso estar flotando en la orilla del muelle? Es decir, si, es gramaticalmente correcto, pero nadie habla así. Entonces por el puro contexto supondríamos que es más sensato que la cuerda sea la flotadora.

Ok suficiente clase de español. Sabemos que para la computadora todo esto van a ser números de cualquier manera.

Imaginemos el significado de cada palabra como la suma de las palabras en la misma oración a las que les pone más atención (para entender su contexto), así es como la auto-atencion está viendo una oración para poder procesarla.

Entonces – la atención se enfoca selectivamente en ciertas palabras para resolver una pregunta externa. ¿De qué ciudad estamos hablando? En este ejemplo – mientras más importante sea una palabra para resolver el ejemplo, más atención le va a prestar.

La Auto-Atencion, por otro lado, también va a tomar en cuenta la relación entre las palabras dentro de la misma oración.

En el modelo de Transformers, estas capas de Multi-Head Attention es donde sucede la computación de atención.

Muchachos, vamos viendo cómo funcionan.

En la primera parte de este bloque están tres capas lineales – una capa lineal es simplemente una capa Densa completamente conectada sin función de activación.

Sirven para lo que ya saben – mapean las entradas a las salidas y cambian las dimensiones de las entradas.

Ok, entonces viene nuestro embedding (palabra + posición) y entra directo a nuestra capa lineal. Cada una de las 3 dimensiones del vector de embedding va a entrar en las 2 neuronas de la capa lineal como entrada y checa como redujimos la entrada de 3 a 2 elementos.

¿Por qué harías tus capas lineales con menos neuronas que dimensiones en tu embedding de entrada? Es simplemente para acelerar el computo, en caso de ser necesario.

Ok, entonces esta parte funciona igual que una red neuronal densa, completamente conectada. ¿Qué sigue?

Vamos enfocándonos en el hecho de que este bloque tiene no uno, sino 3 capas lineales separadas de entrada.

Cada una de estas tiene su propia función especial – Una es la de Query (búsqueda), otra es la de clave y otra es la de valor.

Los nombres de estas secciones vienen de cómo funcionan los sistemas de búsqueda. Imaginemos el ejemplo de buscar una persona en Instagram.

Imaginemos que el algoritmo es muy sencillo – escribes el nombre de tu victima Leonel Messi, por ejemplo, y esta será la Query o búsqueda que estás haciendo. Instagram va a buscar en toda su base de datos el nombre de @leomessi entre todos los nombres de gente que tiene disponibles.

Estos nombres van a ser las claves. Para computar los mejores resultados va a tener que calcular alguna clase de similitud entre tu Query y las claves correspondientes una vez que la clave más similar haya sido encontrada.

Entonces esperar ver que obviamente, el auténtico Leonel Messi tenga el valor más alto en este cálculo, pero también habrá candidatos como @leomessi\_.4ever, leomessi\_supporters\_, leomessi\_.fanp, leomessifans10mex, leoeemessi y así que sacaran puntajes muy altos.

La cuenta de @cristiano sacara valores extraordinariamente bajos – nada que ver con su desempeño futbolístico, es solo que no se parecen nada los nombres.

Entonces, este modelo regresa el mejor perfil según lo que hayas buscado.

Nota como el valor de similitud puede verse como un análogo a la atención en este caso, nuestro modelo solo nos entregara los mejores resultados poniéndole atención al nombre de usuario más similar a nuestra búsqueda.

Ok genial – pero entonces ¿cómo se va a computar la similitud entre una Query (búsqueda) y las claves? La verdad hasta ahorita obviamos ese cálculo.

Vamos a usar un método que se llama similitud de coseno, que es una manera genial para computar la similitud entre 2 vectores.

Al final de cuenta, recordemos que lo que entra son embeddings que están dados en forma de vector. La similitud de coseno va de -1 a 1, donde el 1 significa que los 2 vectores apuntan en la exacta misma dirección, y la de -1 significa que apuntan en la dirección exactamente opuesta.

Si los 2 vectores apuntan en la exacta misma dirección, el ángulo entre ellos va a ser cero, y si tu recuerdas tu trigonometría de prepa, coseno de 0 es igual a uno. ósea, ambos vectores tienen la máxima similitud posible.

Ahora la similitud de coseno se reduce cuando los vectores se empiezan a separar, hasta que la máxima disparidad ocurre cuando están apuntando en direcciones opuestas, en este caso un ángulo de 180. Coseno de 180, es -1.

Para evitarnos hacer cálculos geométricos complicados, podemos también calcular la similitud de coseno entre 2 vectores calculando el producto punto de los 2 y luego dividiendo entre sus magnitudes. así que nuestra ecuación nos quedaría así.

Ahora, si vas a computar la similitud entre una matriz de elementos en vez de vectores, vamos a tener que transponer la segunda matriz para evitar conflictos en temas de las dimensiones al momento de la multiplicación.

Vamos metiendo nuestras queries y nuestras claves en nuestra ecuación.

Ok, ¿y cómo se relaciona esto con nuestra capa de atención, y que deberíamos ajustar a nuestras capas de query, clave y valores? Bueno a la capa de query le alimentamos nuestros embeddings de posición + palabra.

Luego hacemos 2 copias más de embeddings y le alimentamos los mismos a las capas de clave y valor.

A ver a ver a ver, espérenme – van a decir los que han estado poniendo atención. ¿Qué no las queries y las claves y los valores significaban cosas diferentes en el ejemplo que les di de Instagram?

Yes, correcto, pero vamos a usar las mismas entradas para estos valores por el tema de la auto atención. Vamos a tomar 3 copias del embedding y ponerlas aquí al lado, luego las pasaremos por cada una de las capas lineales y todo lo que eso significa es que multiplicamos nuestras capas de embedding con los pesos de esas capas lineales.

Nota que cada capa lineal tiene su propio set de pesos. Ya que la multiplicación de matriz requiere ciertas dimensiones, vamos a tener que transponer nuestras matrices de embedding de manera acorde.

Después de la multiplicación cada capa lineal saca una nueva matriz y estas se llaman, imaginativamente, las matrices de query, de clave y de valor.

Vamos a enfocarnos solo en las matrices de query y de clave, por ahora – recuerda como solo las queries y las claves fueron usadas en nuestro ejemplo para computar la similitud.

Entonces hacemos primero un simple producto punto entre nuestra query y la transposición de nuestra matriz de clave – la salida de este producto puede ser llamado un filtro de atención. Y es muy importante, así que vamos viendo cómo entender sus contenidos.

Al principio del proceso de entrenamiento, los contenidos del filtro de atención son básicamente números aleatorios, pero una vez que los procesos de entrenamiento están hechos, agarran valores más significativos.

Si revisas de cerca los valores dentro de esta matriz, son de hecho puntajes d atención. Por ejemplo, consideremos la fila correspondiendo a la palabra “Friends” o amigos. La cantidad más alta de atención que una palabra pone es generalmente a sí misma, así que es lo más similar a ella sola.

El siguiente punto de atención más alto dado a la palabra que tiene el siguiente más similar a ella misma. En este caso, como solo dimos la entrada “keep your Friends close” lo más cercano seria close (ósea, cercano pues).

Finalmente tenemos que escalar nuestros puntajes de atención, los autores del paper dividen el puntaje de atención entre la dimensión del vector clave, en nuestro ejemplo ese es 7. Finalmente, aplastamos nuestros puntajes de atención entre valores de 0 y 1, usando una función sigmoide, y obtenemos nuestro filtro de atención final.

OK, vamos recapitulando lo que vimos hasta ahorita. Hicimos 3 copias del embedding, se las pasamos a la capa lineal de valor y obtuvimos una matriz de valor, y las otras embeddings las pasamos por las capas lineales de query y de clave y escalamos los resultados para obtener un filtro de atención.

Ok, ahora tenemos nuestra matriz de valor original, que representa bastante bien la información del embedding original, esto porque no las alteramos mucho excepto por pasarlas a través de una sola capa lineal.

Por otro lado, tenemos nuestro filtro de atención con puntajes de atención que fueron computados usando producto punto entre las matrices de query y clave. ¿Excelente, pero por que hicimos todo esto? ¿Para qué sirve este filtro de atención y la matriz de valor?

Bueno, aunque estamos lidiando con NLP es mucho más fácil entender la intuición detrás de esto usando visión por computadora.

Supongamos que estas bajando una colina que no conoces a toda velocidad en tu bicicleta de montaña cuando de repente ves estas banderas rojas y letrero amarillo en el camino frente a ti.

Si te esperas que tu cerebro procese toda la imagen, píxel por pixel cuadro por cuadro, vas a acabar en el fondo de un barranquito con un par de fracturas expuestas y una bici muy cara que ya no te sirve.

Por suerte nuestros cerebros funcionan mucho más rápido que eso y no se esperan a ver la imagen completa para procesarla – el cerebro simplemente dice, banderas rojas, letreros amarillo enorme, me paro o me hago al lado, y hago preguntas después.

Así que nuestro sistema de visión es en realidad una imagen de fondo, a la cual no le ponemos mucha atención, y la imagen en la cual estamos enfocados en el momento y en ese contexto, que está ocupando casi todos nuestros recursos cerebrales.

La imagen final que vemos en la mente es una combinación de ambos cuadros, el filtro de atención y la imagen original: si multiplicas el píxel del filtro de atención con la imagen original, obtienes la imagen filtrada con todos los detalles innecesarios eliminados.

De la misma manera, cuando multiplicamos nuestro filtro de atención por la matriz de valor obtenemos una matriz de valor que asigna alto enfoque a las variables que son más importantes, y esta matriz filtrada de valor es la salida final de nuestra capa de atención multi cabeza.

Bueno, casi, hay un pequeño detalle que agregar – ¿recuerdan que dijimos que la atención multicabeza es un mounstruote como la hidra de 3 cabezas? Bueno, apenas llevamos una cabeza, hay 2 más que necesitamos cubrir.

¿Por qué necesitamos más de una cabeza? Vamos de regreso al ejemplo de la bicicleta de montaña. El primer filtro nos había ayudado a enfocarnos en las señales de peligro – banderas rojas y letrero amarillo diciendo “barranco”.

Pero si eres un ciclista experimentado, podrías buscar alguna posible salida, o alguna indicación que existe una rampa para brincar con un poco más de seguridad (como suele haber en esta clase de pistas).

De la misma manera, los Transformers no aprenden solamente un filtro de atención, aprenden varios, cada uno enfocándose en un fenómeno lingüístico diferente. Entonces, cada cabeza de atención saca su propio filtro de atención, que a su vez saca su propia matriz filtrada de valor, cada una enfocándose en diferentes combinaciones de características lingüísticas.

En el paper original de Transformers, los autores usaron un total de 8 cabezas de atención, sin embargo, nosotros usaremos 3 para el ejemplo.

Ok, entonces lo que sigue es simplemente concatenarlas juntas ya que no queremos que este vector se haga más y más largo con cada cabeza siendo usad, y lo pasaremos a través de un mapa lineal para reducirlo a 7x3 y esta es la salida final de la capa de atención multi cabeza.

¡Excelente! En el último video vamos a discutir los componentes que quedan, así como la parte del decodificador.

### **El decodificador**

Ok muchachos, ya discutimos los embeddings de posición, ya discutimos los conceptos de atención, auto atención y atención multi cabeza – ahora solo queda revisar el resto del modelo.

El resto del modelo este compuesto de elementos que ya conocen – conexiones de skip (residuales), normalización de capa, módulos de decodificación y capas de softmax. Vamos a darle una repasada a todos estos componentes en el contexto de los Transformers. Al final vamos a ver la parte que es nueva para ustedes: atención enmascarada.

Vamos entrando directo a las conexiones skip o residuales. Recordemos que estas conexiones sirven 2 funciones: preservar la información y resolver el problema de los gradientes que se desvanecen.

En los Transformers nos importan para el tema de preservar la información. Tenemos nuestra red neuronal compuesta de un montón de neuronas durante la propagación hacia adelante. Obviamente la info que entra se modifica considerablemente, y para el tiempo que alcanzan la última capa, esto va a resultar en la pérdida de información útil que estaba presente al principio de la corrida.

Por ejemplo, estos datos iniciales traen la información de embedding posicional, que tal vez se olviden para cuando llegas a la última capa de neuronas - ahora la pregunta es, ¿cómo nos aseguramos de que la info original no se pierde?

Una solución es agregar conexiones residuales, que esencialmente son carreteras de información que le dan la vuelta a toda a la red y alimentan la información directo a capas más profundas. Cuando simplemente agregamos estas 2 piezas de información, y obtenemos nuestra capa final de salida, esto ayuda a la capa más profunda a no olvidar información más importante que estuvo presente al principio del sistema.

Y de aquí en adelante la propagación hacia delante continua como de costumbre.

Así que tomar una conexión residual es como tomar un atajo, saltándose a todos y luego volviendo a la pista de carreras.

OK, ahora que ya tenemos la salida de la capa de embedding que usa las conexiones residuales, vamos a la siguiente capa. También tenemos la salida del componente de atención multicabeza que, a su vez, pasa a la siguiente capa.

así que, que es esta siguiente capa? Bueno podemos llamarlo la capa de suma y normalización, por que suma las 2 entradas que se alimentan a esta capa – esa parte esta fácil.

La otra parte es que aplica un batch norm a nuestra salida para fines de regularización. Ustedes ya conocen batch norm, pero vamos viendo como aplica en un transformer específicamente.

Vamos haciendo zoom a esta salida sumada para entender esto mejor. Como puedes ver, esta afila corresponde a una palabra en la oración y cada columna variables.

La normalización de capa lo que hará es estandarizar las activaciones de neuronas a lo largo del eje de las variables. Para hacer eso tomamos las activaciones de las variables y les restamos su promedio y luego dividimos por la desviación estándar. Finalmente agregamos un valor pequeño al denominador solo para asegurarnos que nunca acabe siendo cero.

Y listo, todos los valores de la matriz ya fueron normalizados – prosigamos.

En esta fila tenemos un montón de capas lineales con activaciones entre ellas. Ya conocemos las capas lineales, solo que esta vez sí hay una función explicita de activación, RELU. Ya si no la conocen a estas alturas del campeonato cuestionen que han hecho las últimas 40 horas de curso.

Y con esto pues ya vimos todos los elementos del codificador.

Ahora sí, vamos pasando al decodificador. Trabaja ligeramente diferente, así que pongan atención.

El codificador toma el texto de entrada y lo convierte en alguna clase de representación vectorizada. El decodificador luego toma esta representación y la convierte en texto nuevo, ahora una principal diferencia entre el codificador y el decodificador es que mientras el codificador toma solo una entrada que es el texto fuente, el decodificador toma dos, el primero siendo la salida del codificador y la segunda siendo el texto de salida que ha sido generado hasta ahora.

Vamos comenzando con esta primera entrada después de obtener la salida del codificador. La separamos en 2 copias, estas son la de query y la de clave.

Por ahora, dejemos estas matrices aquí y enfoquémonos en el decodificador, que principalmente lidia con el texto generado. Nosotros suponemos que la primera palabra que el decodificador genera es un token especial indicando el inicio de la oración generada.

Alimentamos este token como la primera entrada al decodificador, de ahí viaja a través de la capa de embedding de la salida.

Lo que lo convierte en un vector de embedding, y luego agregamos información posicional y lo pasamos a la capa de atención multicapa. Ignoremos esto de la máscara por ahora, volveremos a este tema después.

La salida de la capa de atención multi cabeza se va a la capa de suma y normalización, y esto nos da nuestra matriz de valores. Finalmente, mandamos nuestra matriz de valores a la segunda capa de la atención multi-cabeza.

Ahora, si nos asomamos a esta capa de atención, vas a ver que de hecho toma tres entradas, la primeria siendo la matriz de query y de clave que vinieron del codificador y la segunda siendo la matriz de valor que viene de la secuencia de texto previamente generada.

El módulo de atención multicapa agarra estas tres entradas y luego trabaja de la exacta misma manera que vimos en el video anterior.

Las salidas de la capa de atención multi cabeza fluye hacia adelante. Vamos haciendo hincapié en ella ultima capa lineal, ya que esta es medio importante – les voy a explicar cuántas neuronas tendremos al final.

Si han estado poniendo atención al curso, ya deberían de saberlo.

La última capa lineal contiene un montón de neuronas completamente conectadas, ahora el tamaño de esta capa depende en el número de clases. Si tienes un clasificador que te está distinguiendo peras de manzanas, entonces solo tienes 2 clases, solo le pones 2 neuronas a la capa final.

Si tienes un clasificador con peras, manzanas y naranjas, tienes 3 clases, tienes 3 neuronas.

Así que el tamaño de la capa final es el tamaño del vocabulario total que estamos usando. Si lo piensan, de cierta forma la generación de texto es simplemente un ejercicio de clasificación, donde cada palabra podría ser considerada su propia clase.

Ósea, esto también funcionaria con una simple regresión logística.

Lol – si no entendieron que eso era un chiste están en problemas.

Si nuestro texto es “keep your Friends close” entonces el clasificador va a tener que asignarle probabilidades a cada una de las n-palabras en su vocabulario, y luego escogemos la palabra con la probabilidad más alta de seguir generando.

La entrada a la capa final no es texto crudo, pero la salida puede venir de la capa de suma y normalización.

Si mandamos esta matriz, así como esta, la salida será tal que cada unidad en la capa lineal final saca un vector, ahora el problema es que no queremos vectores lo que queremos son puntajes de probabilidad para cada palabra, así que podemos escoger la palabra con el puntaje máximo.

Para lograr eso, primero debemos de aplanar nuestra matriz en una sola fila que luego concatenamos. Luego las pasamos a la capa lineal. Esta información es pasada por cada dimensión de entrada siendo multiplicada por los parámetros de la siguiente capa, y luego siendo sumada junta.

Como resultado de esto obtenemos un solo puntaje para cada palabra, llamamos estos puntajes crudos los logits y luego pasamos estos puntajes a la capa de softmax. En esta capa softmax los convierte en probabilidades.

Finalmente podemos escoger la palabra con la probabilidad más grande para generación. Ahora que entiendes la primera pasada, vamos viendo cómo funciona el proceso de decodificación en total.

Hasta ahorita vimos que el proceso de decodificación consume la salida del codificador. También nos recibe la primera palabra generada, que en este caso es un token especial de inicio de oración indicando el inicio del texto generado. Este token entra a la capa de embedding del texto de salida y se le agrega la información posicional.

Finalmente, pasamos eta info a las capas que quedan y tenemos la siguiente palabra generada.

Vamos a seguir los mismos pasos para esta palabra recién creada, excepto que ahora el decodificador consume ambas, la primera y segunda palabra, y cuando estas obtenemos una tercera palabra en la secuencia. Este proceso continuo hasta que el decodificador genere un token especial de final. ¡Y listo! así funciona el decodificador en una pasada completa.

Ahora sí, vamos al último elemento. Atención enmascarada

### **Atención Enmascarada.**

El principal uso de los módulos de enmascaramiento (o masking, como les diremos de ahora en adelante) sale a brillar durante la fase de entrenamiento.

Al contrario de las pruebas/producción, donde no conocemos las respuestas antes, durante la fase de entrenamiento el modelo recibe ambos los diálogos fuente junto con como completar estos diálogos (ósea los diálogos objetivo).

Esto le permite al modelo aprender de sus errores, vamos a escoger un dialogo de muestra para hacer esto más concreto.

Que les parece este del silencio de los inocentes: “A census taker once tried to test me, I ate his liver with some fava beans a nice chianti”.

Pero bueno el punto es que durante el entrenamiento solo le daríamos la primera parte de la línea para el codificador: A census taker once tried to test me.

Como esto es entrenamiento, el decodificador también va a recibir “I ate his liver with some fava beans and a nice chianti”, que en este caso es el dialogo objetivo.

Dicho eso, el dialogo objetivo va a venir con mascara puesta. Esto por la exacta misma razón por la cual no estudias para un examen con las respuestas descubiertas en el quiz de repaso.

Se supone que primero haces tu mejor esfuerzo por recordar y luego revisas si estabas bien o mal. De esta manera es más probable que aprendas de tus errores y se te queden grabadas las respuestas correctas.

De manera similar, después de que metemos el texto de entrada en el transformer, el decodificador genera su primera predicción. Digamos que esta primera palabra es You. Luego desenmascaramos la respuesta correcta I.

Ahora, en esta etapa de entrenamiento, en vez de pasarle al decodificador sus propias predicciones de regreso (como lo haríamos en pruebas), le pasamos la palabra objetivo actual. Esto se llama “**Teacher forcing**” o Forcejeo pedagógico. Esto ayuda al modelo en cuantificar la diferencia entre los puntajes de la distribución de probabilidades entre las etiquetas verdaderas y las predicciones del modelo.

Una manera de cuantificar esta diferencia es computar la pérdida es usando la función de Perdida de entropía cruzada.

Ok, ya vimos porque necesitamos el masking.

Ahora vamos viendo como se hace.

Si recuerdas, habíamos visto en el video de atención como generar los puntajes de atención. Antes de pasarlos a la capa de softmax, vamos a hacer la operación de masking. así que es Masking? Es simplemente una matriz de filtro donde todas las palabras que vendrán en un futuro reciben un puntaje de -infinito, de esta manera el modelo no las va a pelar.

El resultado final se ve algo como así.

Una vez que pasa el filtro de atención enmascarado a través de la capa de softmax, checa como los infinitos negativos se volvieron 0s. Por ende, cuando predecimos la palabra después de “ate”, modelo le pone atención solo a las palabras anteriores de “ate” y le pone 0 atención a las palabras que la siguen.

Otra visualización es esta:

En el tiempo 0 el modelo le da atención al texto vectorizado de entrada obtenido del codificador, y el token especial de inicio. Ahora digamos que acabo prediciendo la palabra You, entonces desenmascaramos la primera etiqueta de verdad y se la pasamos al decodificador.

Por ende, su próxima predicción que es el paso de tiempo 1, el modelo le pone atención a la salida del encoder, así como la palabra objetivo de verdad del paso previo.

Digamos que ahora acaba prediciendo la palabra “are”. Ahora desenmascaramos el segundo objetivo, y el proceso continuo hasta que se genera el token de finalización. Cuando esto sucede nota como el modelo está poniendo atención al texto de entrada junto con todos los tokens de salida hasta el token final se aparece.

En este ejemplo acaba generando trabalenguas, pero pues para eso es el entrenamiento, para mejorar sobre tus predicciones del pasado. Durante la inferencia, los tokens de verdad objetivos son reemplazados por las predicciones del modelo un paso atrás.

¡Y listo! Ya entendemos cómo funcionan los Transformers.

Ahora, para verlos en acción vamos a ir a este transformer de **HuggingFace**, que están ejecutando un modelo de transformer que se llama GPT2.

Aquí esta lo que el modelo me arrojó.

Si se fijan no es nada que ver con la película, pero gramaticalmente tiene sentido, y es una frase que podrías escuchar a alguien decir en la vida real.

¡Y listo! Con esto ya entiendes los Transformers, que son una de las arquitecturas más poderosas para el manejo de texto hoy por hoy.

Te recomiendo que te familiarices con los modelos de producción más famosos de Transformers GPT3 y BERT, que actualmente son los reyes en procesamiento de lenguaje natural.