I. Taller 8 – Demanda de bicicletas (RNN)

A. 1

Ambos modelos comparten la misma arquitectura básica: una capa recurrente seguida de una capa densa.

En el caso del modelo Seq-to-Vec, la cantidad total de parámetros cuando se utiliza una capa recurrente SimpleRNN, se obtiene sumando los siguientes componentes:

1-Los pesos que conectan las entradas (features) con la capa recurrente. $(\#Features * n_{neuronasRecurrente})$

2-Los parámetros que conectan cada neurona recurrente con las demás en la misma capa (retroalimentación). $(n_{neuronasRecurrente} * n_{neuronasRecurrente})$

3-El término de bias asociado a la capa recurrente. $(n_{neuronasRecurrente})$

4-Los pesos que conectan la salida de la capa recurrente con la capa densa. $(n_{neuronasRecurrente} * n_{neuronasDensa})$

5-El bias de la capa densa. $(n_{neuronasDensa})$

En este caso, con 57 features, 64 neuronas recurrentes, 1 capa densa de 1 neurona, el calculo queda

$$Parametros_{SimpleRNN} = 57*64+64*64+64+64*1+1 = 7873$$

Donde 7808 parámetros son por la capa recurrente y 65 por la capa densa.

En el caso de utilizar una capa LSTM en Seq-to-Vec, el cálculo de parámetros es distinto al de una capa recurrente simple, ya que cada celda LSTM incorpora cuatro compuertas que controlan el flujo de información: Las compuertas f(t) (forget gate), i(t) (input gate)y o(t) (output gate) actuan como selectores modulando el flujo de memoria y la compuerta g(t), que transporta la información de estado.

Cada una de estas compuertas posee su propio conjunto de pesos y bias, tanto para las conexiones con las entradas como para las conexiones internas entre unidades recurrentes. Por esta razón, todos los parámetros asociados a la capa recurrente se multiplican por 4.

Por lo tanto, el cálculo de parámetros LSTM es:

$$Param_{LSTM} = 4.(Param_{Recurrente}) + Param_{Densa}$$

$$Param_{LSTM} = 4.(57 * 64 + 64 * 64 + 64) + 64 * 1 + 1$$

$$= (4 \cdot 7808) + 65 = 32197$$

.

La principal diferencia entre el modelo seq-to-vec y el seq-to-seq es la forma de la salida, en lugar de predecir un único valor, el modelo seq-to-seq genera una secuencia completa como salida.

En el modelo seq-to-vec, la RNN procesaba una secuencia de 24 horas pero solo predecía una hora futura. Esto implica que mucha de la información temporal aprendida se utiliza para generar una única predicción, lo cual puede limitar su capacidad de generalización.

En cambio, el modelo seq-to-seq aprovecha mejor la estructura temporal: cada paso de la RNN produce una salida correspondiente, lo que permite que cada entrada tiene una salida esperada. Esto tiende a reducir el sesgo y mejora la capacidad de generalización del modelo.

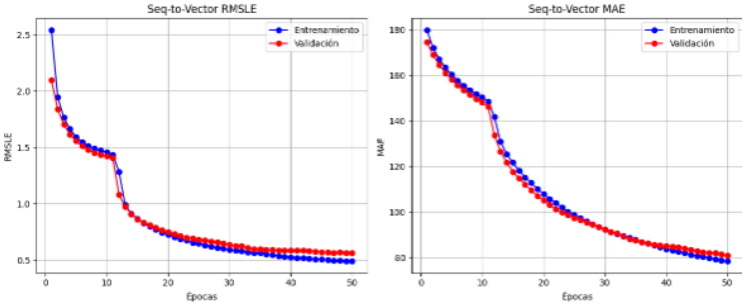

Fig. 1: Comparación de MAE y RMSLE para seq-to-vec

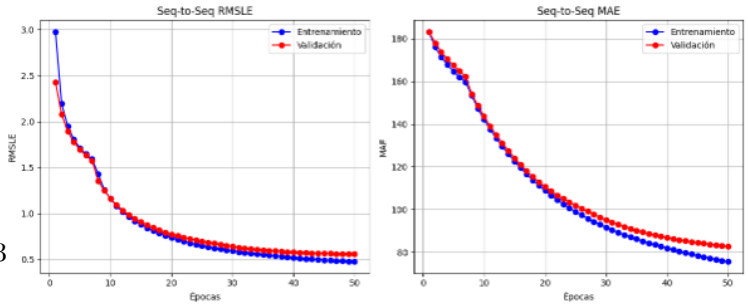

Fig. 2: Comparaciónn de MAE y RMSLE para seq-to-seq

La cantidad de parámetros en la capa recurrente es la misma que en el modelo seq-to-vec, ya que depende únicamente del número de unidades, la dimensionalidad de entrada y el tipo de celda recurrente utilizada. Lo que cambia es cómo se utiliza la salida durante el entrenamiento.

Por lo tanto en seq-to-seq, $Param_{SimpleRNN} = 7873$ y $Param_{LSTM} = 31297$

B. 2

Las consideraciones que deben tenerse en cuenta para poder comparar ambos modelos son principalmente sobre la forma de sus salidas. El modelo seq-to-vector, dado un batch de secuencias de entrada, devuelve una única predicción escalar por secuencia. En cambio, el modelo seq-to-seq genera una secuencia de predicciones, una por cada paso temporal de la entrada (seq_length valores).

Por ejemplo, el modelo seq-to-vector recibe como entrada las ventas de las últimas 24 horas y predecir la cantidad de ventas en la próxima hora. Mientras que el modelo seq-to-seq usaría las mismas 24 horas de entrada pero devolvería una predicción para cada una de las próximas 24 horas.

Esto implica que para poder comparar métricas hay que hacer una modificacion a la metrica MAE para que en vez que tome como valor los errores de las 24hs, tome solamente la última.

El rendimiento para ambos modelos usando una Simple RNN se ven en al Figura 2 y Figura 1

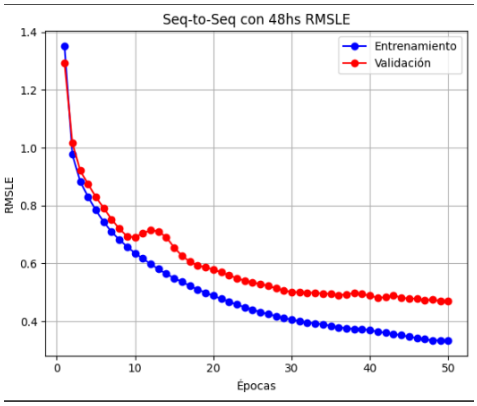

Fig. 3: RMSLE para modelo seq-to-seq con secuencias de 48hs (sin dropout)

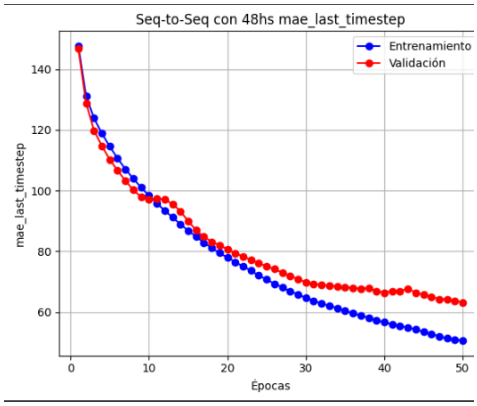

Fig. 4: MAE para modelo seq-to-seq con secuencias de 48hs (sin dropout)

C.

Una de las estrategias que exploré para mejorar los resultados del modelo seq-to-seq fue incrementar la longitud de la secuencia de entrada (seq_length) de 24 a 48 horas. Es decir, en lugar de predecir las próximas 24 horas a partir de las 24 horas anteriores, pasé a predecir las próximas 24 horas utilizando como entrada las últimas 48 horas de datos históricos.

En problemas de series temporales como la predicción de demanda de bicicletas, es razonable pensar que un mayor contexto histórico puede ayudar al modelo a capturar mejor tendencias y patrones semanales. Dado que el modelo seq-to-seq debe generar una predicción por cada hora futura, contar con más información previa puede ser clave para una mejor generalización.

Para este generar un mejor modelo, utilicé celdas LSTM entrenando una red con 64 neuronas LSTM, seguidas de una capa Dense con una sola neurona envuelta en TimeDistributed, para poder generar una salida por cada paso de la secuencia. Los resultados de esta primera prueba se pueden ver en la Figura 4 y 3 :

Como se puede ver en la figura, a partir de la época 30 la curva de entrenamiento sigue mejorando, pero la de validación

```
model_lstm_dropout = Sequential([
    LSTM(64, return_sequences=True,dropout=0.2,recurrent_dropout=0.2),
    TimeDistributed(Dense(1, activation='relu'))
])
model_lstm_dropout.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
    loss=rmsle,
    metrics=[mae_last_timestep]
)
```

Fig. 5: Arquitectura final del modelo con dropout

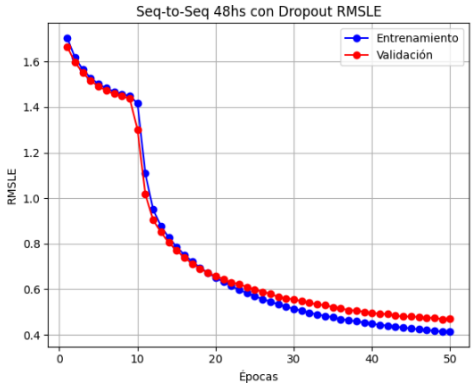

Fig. 6: RMSLE para modelo seq-to-seq con secuencias de 48hs (con dropout)

se mantiene constante. Esto me llevó a pensar que el modelo estaba empezando a sobreajustarse un poco.

Para tratar de evitar esto, le agregué dropout al modelo: tanto sobre los pesos normales de las neuronas como sobre los pesos recurrentes.

El modelo final quedó como se muestra en la Figura 5:

Y los nuevos resultados con dropout aplicado se pueden ver en la Figura 7 y 6:

II. Taller 9 – Crítica de películas (RNN)

A. 1

La capa TextVectorization de Keras transforma texto en una secuencia de números enteros (vectores). Entre los parámetros configurables más relevantes se encuentran: Max Tokens , que

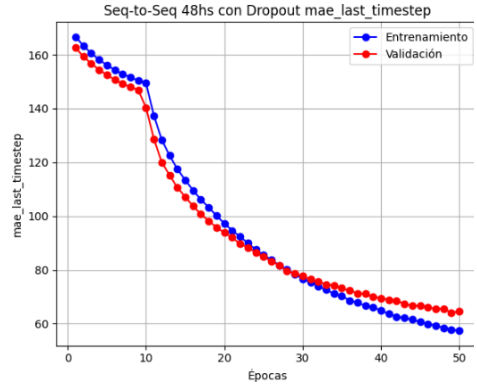

Fig. 7: MAE para modelo seq-to-seq con secuencias de 48hs (con dropout)

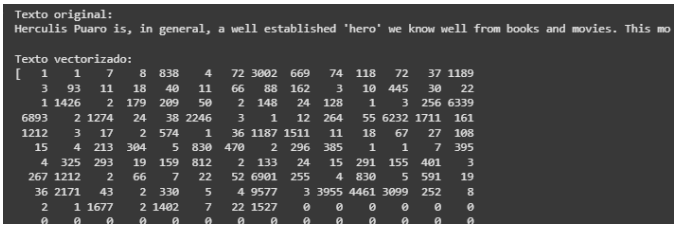

Fig. 8

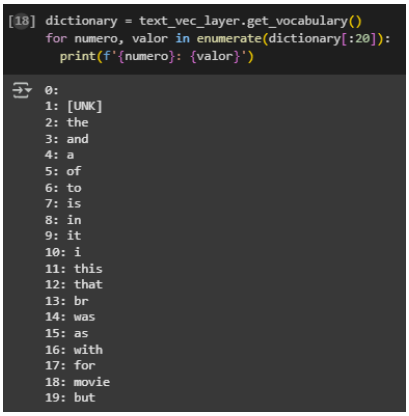

Fig. 9: Diccionario

define la cantidad máxima de palabras que puede contener el vocabulario. El parametro Split, que indica cómo separar el texto: por ejemplo, puede tokenizar por palabras o por caracteres.

En la Figura 8, se observa un ejemplo: se toma una review de un batch del conjunto de entrenamiento (parte superior de la figura) y se le aplica la capa TextVectorization, obteniéndose una secuencia de enteros (parte inferior).

Cada palabra del texto es mapeada a un número entero según su frecuencia en el corpus de entrenamiento. Las palabras más frecuentes reciben números más bajos. Las palabras que no aparecen en el vocabulario entrenado (por ejemplo, aquellas que solo aparecen una vez o no alcanzan un umbral mínimo de frecuencia) son representadas por el token especial [UNK], que suele tener el valor 1.

En el ejemplo de la Figura 8, las dos primeras palabras del texto son ”Herculis Puaro”. Al ser un nombre propio poco común, probablemente no aparezca en otras reviews, por lo que ambas son mapeadas al token desconocido (1). En cambio, la siguiente palabra, ”is”, es muy común en inglés, por lo que recibe un número bajo dentro del vocabulario.

Esto se puede confirmar observando las primeras posiciones del vocabulario, que se muestra en la Figura 9.

B. 2

Las reviews no tienen todas la misma longitud, así que cuando se arma un batch, las que son más cortas se rellenan con ceros al final (zero-padding) para que todas tengan el mismo tamaño. Esto es necesario porque las RNN esperan que todos los elementos del batch tengan la misma forma.

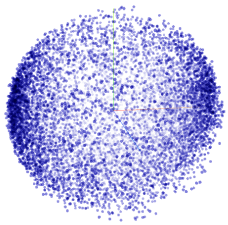

Fig. 10: Embedding Entrenado

Fig. 11: Embedding preentrenado de GloVe

Aunque se agreguen ceros, las redes recurrentes pueden manejarlos sin problema, porque se puede indicar que esos ceros se ignoren durante el entrenamiento. Esto se hace usando una capa de Masking.

C. 3

La capa de Embedding toma cada palabra tokenizada (representada como un número entero) y la transforma en un vector de dimensión fija. Estos vectores capturan relaciones semánticas entre las palabras: aquellas con significados similares tienden a tener vectores cercanos en el espacio vectorial.

Una ventaja de entrenar una capa de embedding desde cero es que se adapta mejor al lenguaje específico del problema. Si el vocabulario contiene jerga o palabras poco comunes, la red puede aprender representaciones adecuadas directamente de los datos.

Usar un embedding ya preentrenado te permite aprovechar vectores aprendidos sobre conjuntos grandes (IMDb), tener un vocabulario más amplio, incluyendo palabras que no aparezcan en tu conjunto de entrenamiento y además reduce el riesgo de sobreajuste. También te ahorra tiempo y datos, ya que no necesitas entrenar los embeddings desde cero.

La comparación entre las gráficas de los embeddings muestra que el embedding entrenado desde cero tiene una distribución más esférica y homogénea, donde se puede distinguir claramente la separación semántica entre las palabras (ver Figura 10).

En cambio, en la Figura 11, donde se visualiza el embedding preentrenado GloVe, la forma es más asimétrica y concentrada en ciertas regiones del espacio. Esto puede indicar que GloVe prioriza ciertas relaciones semánticas globales.

```
model_trainable = tf.keras.Sequential([
    text_vec_layer,
    embedding_layer_f,
    tf.keras.layers.GRU(128, dropout = 0.4),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

model_trainable.compile(loss="binary_crossentropy", optimizer="nadam", metrics=["accuracy"])
```

Fig. 12: Modelo Final

Modelo con Embedding	Total params	Trainable	Optimizer params
Entrenado desde cero	4,137,606	1,379,201	2,758,405
Preentrenado (trainable=True)	6,265,950	2,088,649	4,177,301
Preentrenado (trainable=False)	2,000,200	2,000,200	0

TABLE I: Comparación de cantidad de parámetros según el tipo de embedding utilizado

D.

Un nuevo modelo se intentó con un embedding preentrenado con el parámetro trainable en True, aumentando el tamaño del vocabulario, generando un dataset más limpio, sacándole símbolos propios de HTML, llevando las letras a minúsculas y eliminando signos de puntuación.

Con estos cambios, el modelo queda como en la Figura 5 que llega a una $val_{accuracy} : 0.8820 - val_{loss} : 0.2909$.

Mientras que al usar un embedding entrenado desde cero con una capa GRU regularizandola con un dropout de 0.4, llega a $val_{accuracy} : 0.8990 - val_{loss} : 0.2549$

Un punto a tener en cuenta son la cantidad de parametros mostrada en la Tabla I

Aunque usar una capa de embedding preentrenada puede parecer una mejor idea que entrenarla desde cero, en este caso, al no tratarse de un dataset tan grande ni tan diverso, ambos modelos obtienen resultados similares, e incluso entrenar el embedding desde cero da resultados levemente mejores y utiliza menos parámetros que si se entrena una capa de embedding ya preentrenada.