Segunda Entrega
Taller de Aprendizaje Automático

Instituto de Ingeniería Eléctrica

Federico Bello

28 de septiembre de 2025

Taller 8 - Demanda de bicicletas

1. Utilizando el método *summary* es posible ver la cantidad de parámetros de un modelo en cada capa, haciendo esto se obtienen los parámetros totales de la tabla 1. Para el caso de la capa recurrente simple se precisa un parámetro por entrada, uno por estado y uno de *bias*, esto para cada neurona. Esto puede descomponerse en: $(cant\_neuronas + size\_entrada + 1) * cant\_neuronas$, donde en este caso la cantidad de neuronas es 64 y el tamaño de entrada 57 (cantidad de características), obteniendo que la capa recurrente aporta 7808 parámetros entrenables. A esto, hay que sumarle los parámetros de la capa densa, es decir, un parámetro por neurona (64) mas el termino de *bias*, obteniendo así el resultado de la tabla. Luego, para el caso de la arquitectura LSTM se tiene que cada neurona tiene 4 veces mas parámetros que en el caso de la simple, dado que además de la capa simple tiene otras tres capas, las cuales controlan las compuertas de entrada, olvido y salida. Por lo tanto, la capa recurrente consta de $7808 \times 4 = 31232$ parámetros, que sumados a la capa densa se obtiene el resultado de la figura. Además, observar como la cantidad de parámetros no depende de si utilizar secuencia a vector o secuencia a secuencia, dado que la cantidad de parámetros depende solamente de la cantidad de neuronas, el tipo de neurona utilizado y la cantidad de características.

Modelo \ Arquitectura	Simple	LSTM
seq-to-vector	7873	31297
seq-to-seq	7873	31297

Cuadro 1: Cantidad de parámetros por modelo

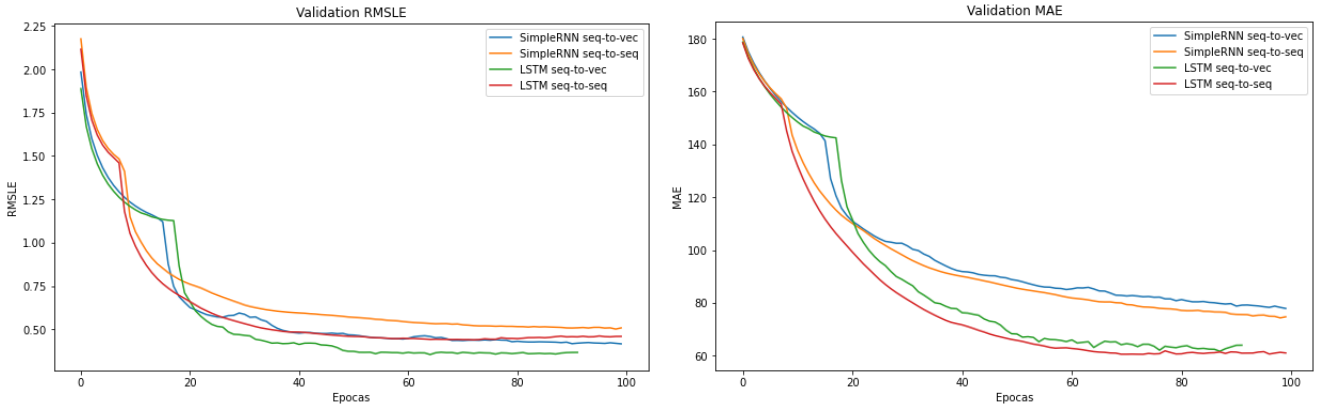
2. Se comenzó probando ambos modelos propuestos, los resultados de los experimentos se pueden ver en la tabla 3. Los mismos son los resultados luego de haber entrenado el modelo durante 50 épocas, utilizando optimizador *Adam* con un *learning-rate* de 1×10^{-3}

Modelo \ Métrica	RMSLE Entrenamiento	RMSLE Validación	MAE Entrenamiento	MAE Validación
Naive forecasting	0.65	0.65	64.5	66.1
seq-to-vec SimpleRNN	0.42	0.46	82.9	88.0
seq-to-seq SimpleRNN	0.50	0.58	78.5	87.1
seq-to-vec LSTM	0.33	0.41	63.8	74.4
seq-to-seq LSTM	0.36	0.46	53.7	67.8
Personalizado	0.35	0.42	43	59.5

Cuadro 2: Desempeño de distintos modelos

Un comentario inicial es que en algunos casos entrenar durante 50 épocas no pareció haber sido suficiente para sacar el máximo provecho sobre el modelo, dado que tanto el desempeño en entrenamiento como en validación seguía mejorando. Sin embargo, en el caso de los modelos LSTM no fue así, comenzándose a sobre-ajustar en las ultimas épocas del entrenamiento. Estas ultimas dos observaciones pueden verse en la figura 1, donde se ve como para las neuronas LSTM ambas métricas se estancan en una cantidad de iteraciones menor, por mas que no

comiencen a empeorar. También se ve como como el modelo *seq-to-vec* con neuronas LSTM consigue un muy buen desempeño en unas pocas épocas. Esto, es indicativo de que incrementar la complejidad del modelo, implementando una regularización y entrenando el modelo durante mas épocas implementando un *EarlyStoppping* puede ser una buena alternativa para mejorar el resultado. Analizando la tabla y las gráficas se pueden extraer otras conclusiones interesantes. El resultado base de *Naive Forecasting*, es decir, predecir el valor solamente copiando el valor anterior, obtiene un resultado razonablemente bueno, siendo comparable con ambos resultados de arquitecturas con *SimpleRNN*, incluso obteniendo mejores valores al utilizar MAE como métrica.



(a) Métrica RMSLE **(b)** Métrica MAE

Figura 1: Desempeño de los distintos modelos en conjunto de validación

Además, se observa como utilizar una capa LSTM en lugar de una simple mejora de forma considerable el resultado, y, en particular, logra superar el desempeño del *Naive Forecasting* bajo alguna métrica. Esto no es algo menor, utilizando un modelo relativamente simple se logro mejorar el resultado base. Esta ultima observación es la clave para el próximo modelo propuesto.

3. El mismo se basa en aprovechar la mayor capacidad de expresión de las neuronas tipo LSTM, pero además hacer uso de que ninguno de los modelos anteriores logro terminar de ajustarse a los datos de entrenamiento. La idea es simple, incrementar la complejidad del modelo creando una nueva capa de pocas neuronas y a su vez, incrementar las neuronas de la capa ya existente, compensando este incremento de la complejidad mediante la implementación de *dropout*. Con esto se llega a un modelo con una primer capa LSTM de 16 neuronas, con un valor de *dropout=0.2*, seguido de una capa con 128 neuronas, también LSTM, y *dropout=0.5*. Este modelo obtiene los valores de la ultima entrada de la tabla 3, luego de entrenar por 53 épocas. Las 53 épocas no fueron un numero arbitrario, sino que se implemento un *callback* con *earlystopping* para evitar el sobreajuste. Observar que el modelo se sigue encontrando considerablemente sobreajustado, indicando que es posible regularizarlo un poco mas para mejorar levemente el desempeño. También se probaron algunas arquitecturas diferentes, sin lograr mejores resultados, dando así a mostrar lo complejo que es el problema de predecir una variable en el tiempo. Algunas alternativas que pudieron haber mejorado el resultado pero no se probaron son las de mejorar el preprocesameiento mediante alguna técnica, como puede ser la creación de nuevas columnas.

Taller 9 - *Natural Language Processing (NLP)*

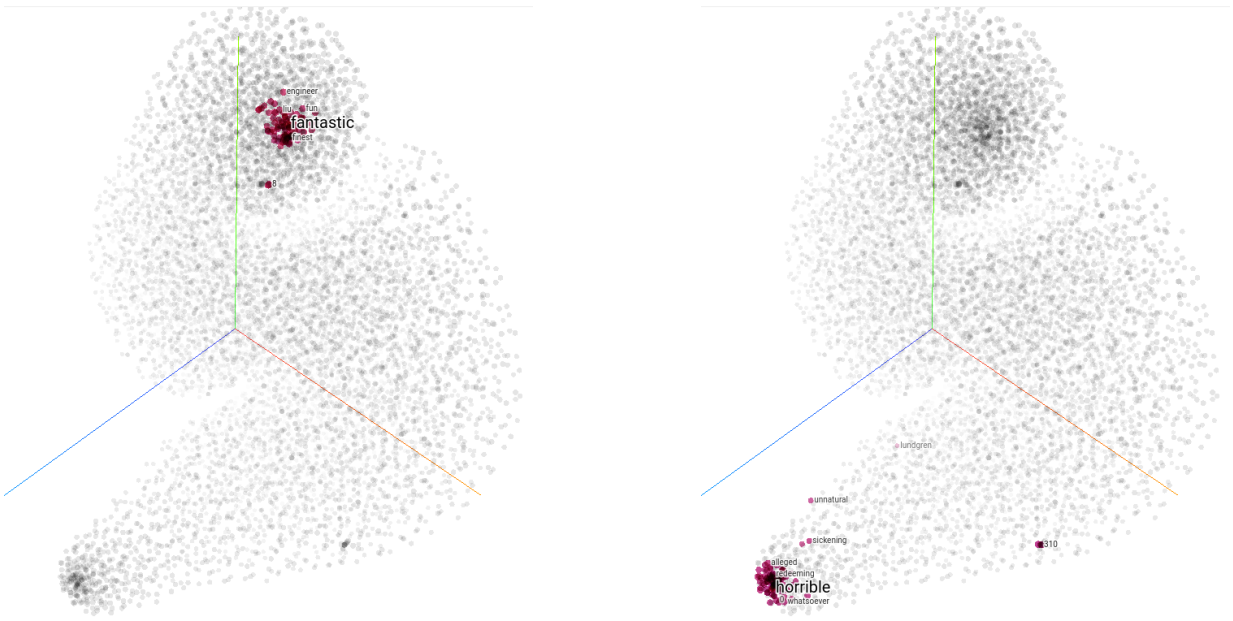
1. La función principal de la capa *TextVectorization* es la de convertir texto en números, consiguiendo de esta forma una representación adecuada para los algoritmos de aprendizaje. Esta idea no es nueva, sino que es similar a utilizar *encoders* para pasar de datos categóricos a datos numéricos. En particular, el texto se podría eventualmente pensar como un dato categórico, solo que uno que tiene una cantidad muy grande de categorías. Esta capa implementa esta codificación de forma secuencial, para que la misma funcione primero es necesario tener un 'diccionario' de palabras, que no es mas que un mapeo de palabras a números. Esta mapeo puede implementarse con esta capa, haciendo uso de del método *adapt*, el cual a cada palabra del conjunto de entrenamiento le asigna un lugar en un *array*, por lo que a esa palabra le corresponde el numero de ese lugar del *array*.

Luego, al recibir un texto el mismo se convierte a *tokens*, que no es mas que cada palabra por separado, por lo que la oración 'Me gusta el aprendizaje automático' se convertiría en ['me', 'gusta', 'el', 'aprendizaje', 'automático']. Por ultimo, realiza el mapeo para cada *token*, es decir, si el diccionario es ['','[UNK]', 'le', 'gusta','aprendizaje','el'] el resultado de la capa sera [1,3,5,4,1]. Lo anterior, si bien es simple, tiene dos detalles. El primero es que la palabra '[UNK]' implica palabras desconocidas, modelando la imposibilidad de predecir todas las palabras que puedan llegar a aparecer. La segunda es que todas las salidas dentro de un *batch* deben tener el mismo largo, por lo que en el caso de que la salida tenga un largo menor se le aplica *zero-padding*, es decir, si el largo para el caso anterior debía ser de 8, entonces la salida correspondiente en realidad sera [1,3,5,4,1,0,0,0], la razón por la cual se reserva el primer lugar en el diccionario.

2. Las redes recurrentes tienen una peculiaridad respecto a otros tipos de redes y es que pueden trabajar con distintos largos de secuencias entre distintos *batches*. Esto, se debe a que el objetivo de las redes recurrentes es procesar entradas de forma secuencial, por lo que en cada instante de tiempo cada neurona procesa simplemente una entrada de la secuencia. Es decir, el tamaño de la matriz de pesos depende solamente de la cantidad de neuronas y de la cantidad de *features* en la entrada, pero no del largo de cada secuencia.

Sin embargo, para poder paralelizar, es necesario que dentro de un *batch* los largos de las secuencias sean iguales. Una solución fácil a esto es aplicar el *zero-padding*, descrito anteriormente, aunque esto tiene la gran desventaja que si la cantidad de ceros es grande (o no tanto) al procesarlos eventualmente se olvida de lo que paso al principio. Una buena alternativa es buscar utilizar batches que tengan aproximadamente el mismo tamaño previo al *zero-padding* (lo cual tiene además la ventaja de acelerar el entrenamiento), aunque esto no siempre es fácil. Otra alternativa mas viable es hacer uso de *masking*, lo que consiste en decirle a la neurona que ignore los *tokens* de valor cero. En todos los modelos hacer uso de esta ultima técnica resulta en una mejora considerable en el desempeño.

3. Existen diversas ventajas y desventajas al momento de entrenar un *embedding* desde cero respecto a utilizar una capa preentrenada. La principal desventaja es que entrenar una capa de este tipo es muy costoso computacionalmente, teniendo que entrenar muchos parámetros distintos. Esto, no solo gasta una cantidad de recursos considerable sino que además es necesario tener un conjunto de datos lo suficientemente grande como para poder llegar a un buen resultado. Sin embargo, tiene la ventaja de poder aprender de datos que son específicos a la tarea en cuestión, logrando potencialmente una capacidad de expresión mayor. Por ejemplo, existe la posibilidad que al momento de utilizar una capa preentrenada una cantidad importante de palabras no estén en el diccionario, limitando así la expresividad del modelo.



(a) Puntos cercanos a 'fantastic'

(b) Puntos cercanos a 'horrible'

Figura 2: *Embedding* entrenado desde cero

4. En las figuras 2 y 3 se observan los *embeddings*, luego de haberlo entrenado desde cero y el preentrenado

con modelo personalizado respectivamente. en los mismos se puede apreciar algo fantástico, las palabras que estan en un extremo de la representación son las que uno calificaría como 'positivas' mientras que las que están en el otro son las 'negativas'. Esto se puede ver en las figuras tomando como ejemplo las palabras *fantastic* y *horrible*. Además, las palabras las cuales toman significados similares en el uso cotidiano están cerca entre si, como puede ser el caso de *fantastic* e *incredible*.

(a) Puntos cercanos a 'fantastic'

(b) Puntos cercanos a 'horrible'

Figura 3: *Embedding* preentrenado con modelo personalizado

En la tabla 3 se observan los distintos desempeños, tanto en términos de la *loss* como de *accuracy*, las primeras tres entradas corresponden a los modelos propuestos en el *notebook* regularizados. Se observa como el *embedding* entrenado desde cero es el que logra un mejor desempeño en validación, aunque el preentrenado entrenable y el no entrenable no quedan atrás. En particular, se observa que estos últimos dos están sobreajustados, indicando que todas las opciones podrían llegar a un desempeño similar si se utilizara una regularización correcta, esto es un indicativo de que la capa de *embedding* no es critica en la clasificaciones, sino que son las otras capas las que aportan realmente a la clasificación.

Métrica / Modelo	*Loss* Entrenamiento	*Loss* Validación	*Accuracy* Entrenamiento	*Accuracy* Validación
Desde cero	0.210	0.264	0.917	0.894
Preentrenado no entrenable	0.215	0.310	0.933	0.8636
Preentrenado entrenable	0.257	0.637	0.943	0.886
Personalizado	0.560	0.674	0.945	0.917

Cuadro 3: Desempeño de distintos modelos

Dados los comentarios anteriores, se busco un modelo personalizado el cual hiciera uso de la capa de *embedding* pero no la tuviera como principal componente, por lo que la misma se regularizo de forma considerable, tanto haciendo uso de la norma l1 como aleatoriamente poniendo en cero alguna de las salidas de la misma (con el método *Dropout)*. Luego, se incluyeron dos capas con neuronas tipo GRU, la primera de 512 neuronas y la segunda de 32, ambas con un *dropout*=0.5. Vale aclarar que esto no fue lo único que se probo. Se hicieron diversos experimentos, tanto probando regularizar principalmente la capa de *embedding* así como principalmente las neuronas. También se probo hacer uso de neuronas LSTM y n-gramas, aunque ninguno de estos cambios resulto ser positivo, por lo que se opto por la arquitectura previamente descripta.

De esta forma, se obtiene el resultado final de 0.674 de *loss* y 0.911 de accuracy en test, logrando así mejorar el resultado del segundo taller.