Deep Learning: Long Short Term Memory(LSTM) № 8

Daniela Pinto Veizaga, dpintove@itam.mx Diego Villa Lizárraga, dvillali@itam.mx

Introducción

En la presente entrega, predecimos los géneros a los que pertenece una película, dada la sinopsis de una película (texto). Para ello, implementamos dos tipos de redes recurrentes: Long Short Term Memory (LSTM) y Gated Recurrent Unit (GRU). Al final, comparamos el desempeño de ambas redes neuronales.

Los datos son obtenidos del siguiente sitio: Kaggle.

Pregunta 1

A pesar de tener muy buena exactitud, la pérdida no bajó de 0.13, y más aún, la salida de la celda anterior muestra que se equivocó al pronosticar las clases positivas, incluso en el set de entrenamiento. ¿Por qué crees que eso sucede?

A pesar de tener muy buena exactitud, la pérdida no bajó de 0.13; más aún, la salida de la celda anterior muestra que se equivocó al pronosticar las clases positivas, incluso en el *set de entrenamiento*. Esto se explica, principalmente, por un problema relativo a la cantidad de datos positivos y datos negativos: hay más datos positivos que datos negativos. En este sentido, el modelo obtienen relativamente buenos resultados prediciendo las etiquetas más frecuentes.

Pregunta 2

¿Por qué crees que este modelo toma mucho más tiempo para entrenarse, en comparación con los modelos que habíamos entrenado anteriormente?

En relación al funcionamiento de las redes recurrentes, como la LSTM, modelan relaciones entre secuencias en lugar de solo entradas ajustadas. Las LSTM contienen celdas que tienen una recurrencia interna (self-loop), además de la recurrencia externa de la RNN. Cada celda tiene las mismas entradas y salidas que una RNN ordinaria, pero tiene mas parámetros y un sistema de compuertas que controla el flujo de información, los cuales vuelven computacionalmente mas costoso su entrenamiento.

En relación al entendimiento del lenguaje, ésta es una tarea compleja, y por lo tanto requiere *embeddings* muy particulares. En particular, para actualizar los pesos de la capa embedding de una red neuronal se realiza propagación hacia atrás. Dado que esta forma de representar vectorialmente las palabras implica el descubrimiento de relaciones secuenciales inherentes a la semántica, se espera que el entrenamiento del modelo sea lento, en la medida que trata de encontrar relaciones con características muy particular y ad-hoc a cada problema.

Pregunta 3

¿Recuerdas cómo funciona el layer Embedding? ¿Qué significan los parámetros de entrada y salida?

De acuerdo con la documentación de Keras, el *embedding layer* es un diccionario que mapea índices de enteros positivos, para cada palabra, hacia vectores densos de tamaño fijo.

El embedding layer de Keras funciona de la siguiente manera: toma como argumentos al menos dos parámetros; el primero, corresponde a 1 + número máximo de índice de palabras; el segundo, corresponde a la dimensionalidad de los embeddings.

- **Input**: tensores de enteros de 2 dimensiones; cada entrada es una secuencia de enteros. Las secuencias que excedan la longitud definida, son truncadas, y las que sean más cortas, se completan con ceros (al inicio) por medio de la operación de padding.
- **Output**: tensores de 3 dimensiones, procesables por RNN, LSTM, capas convolucionales, entre otros.

A continuación, un ejemplo de implementación de una red con layes Embedding, donde el input es el tensor (2000, 16): 1 + número máximo de índice de palabras (primera entrada), con una dimensionalidad de 16.

Listing 1: Red con layers Embedding

```
1
2 Crea una red con layers Embedding, LSTM, Dense
3 model = Sequential()
4 model.add(Embedding(input_dim=2000, output_dim=16))
5 model.add(LSTM(units=16, return_sequences=True))
6 model.add(LSTM(units=32))
7 model.add(Dense(units=64, activation='relu'))
8 model.add(Dense(units=82, activation='sigmoid'))
9 model.summary()
```

Pregunta 4

Diseña y entrena un modelo que minimice la pérdida hasta 0.05 o menos. Además de jugar con los parámetros que ya conocemos, como el número de layers, el número de parámetros en cada layer, loss, optimizer, etc., si quieres también explora otros valores para el número de palabras totales en el diccionario (definidas en el Tokenizer) y los argumentos para el padding.

Logramos obtener un 0.06 de pérdida (modelo 2), sin embargo, notamos que estamos haciendo overfitting (el training loss baja mientras que el validation loss sube).

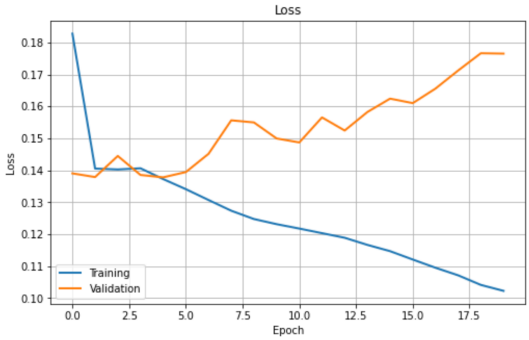
A continuación, se presentan algunos de los modelos con los que obtuvimos mejores resultados bajando la perdida:

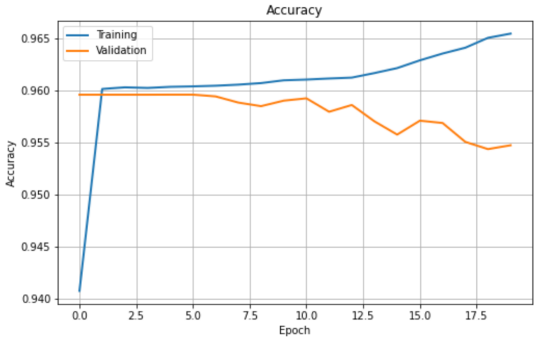
Modelo 1

Layer (type)	Output Shape	Param #
embedding_15 (Embedding)	(None, None, 16)	160000
simple_rnn (SimpleRNN)	(None, None, 32)	1568
dropout_23 (Dropout)	(None, None, 32)	0
simple_rnn_1 (SimpleRNN)	(None, 64)	6208
dropout_24 (Dropout)	(None, 64)	0
dense_17 (Dense)	(None, 64)	4160
dense_18 (Dense)	(None, 82)	5330

Total params: 177,266
Trainable params: 177,266
Non-trainable params: 0

Tokenizer: 10000
Epocas: 20
Batch size: 64
Optimizador: rmsprop
Loss function: Binary Crossentropy
Training Loss: 0.1023
Validation Loss: 0.1765
Training Accuracy: 0.9655
Validation Accuracy: 0.9547





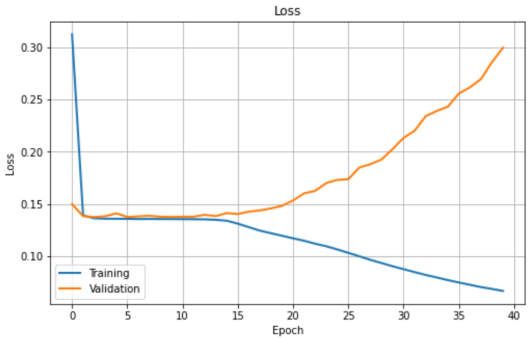
Modelo 2

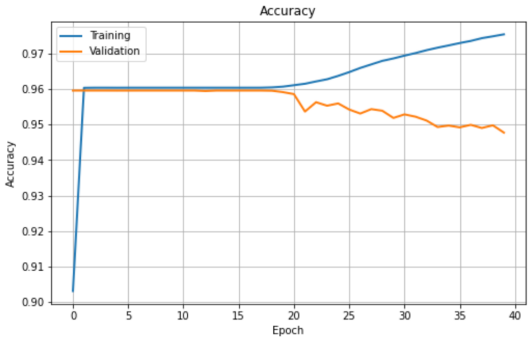
Model: "sequential_16"

Layer (type)	Output Shape	Param #
embedding_16 (Embedding)	(None, None, 16)	160000
simple_rnn_2 (SimpleRNN)	(None, None, 32)	1568
simple_rnn_3 (SimpleRNN)	(None, 64)	6208
dense_19 (Dense)	(None, 64)	4160
dense_20 (Dense)	(None, 82)	5330

Total params: 177,266
Trainable params: 177,266
Non-trainable params: 0

Tokenizer: 10000
Epocas: 40
Batch size: 128
Optimizador: rmsprop
Loss function: Binary Crossentropy
Training Loss: 0.0671
Validation Loss: 0.2995
Training Accuracy: 0.9754
Validation Accuracy: 0.9478

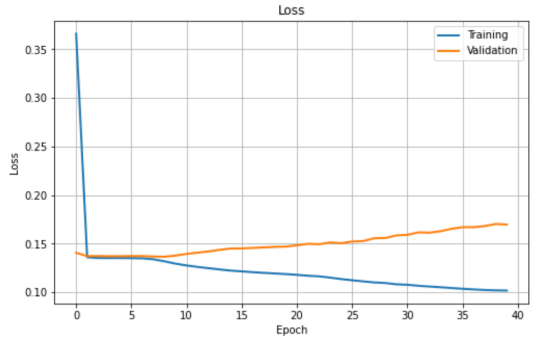


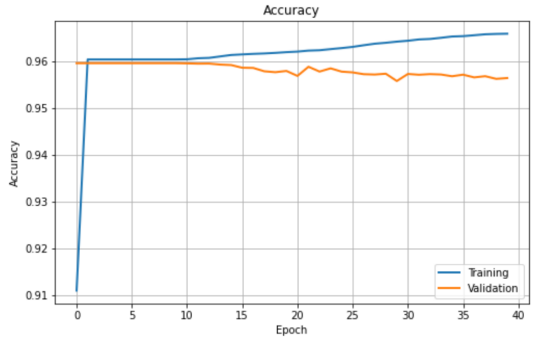


Modelo 3

```
Model: "sequential_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_17 (Embedding)     (None, None, 16)          160000
_________________________________________________________________
lstm_25 (LSTM)               (None, None, 16)          2112
_________________________________________________________________
lstm_26 (LSTM)               (None, 32)                6272
_________________________________________________________________
dense_21 (Dense)             (None, 64)                2112
_________________________________________________________________
dense_22 (Dense)             (None, 82)                5330
=================================================================
Total params: 175,826
Trainable params: 175,826
Non-trainable params: 0
```

```
Tokenizer:  10000
Epocas:  40
Batch size:  128
Optimizador:  adam
Loss function:  Binary Crossentropy
Training Loss:  0.10196227880326497
Validation Loss:  0.16980593685128242
Training Accuracy:  0.96591973
Validation Accuracy:  0.9564306
```





Pregunta 5

Investiga cómo instanciar GRU. Compara RNN vs LSTM vs GRU. Reporta comparación en términos de número de parámetros, tiempo requerido para entrenamiento, y desempeño obtenido.

De acuerdo Chung et al. (2014), la unidad recurrente cerrada (GRU, por sus siglas en inglés), propuesta por Cho et al. (2014), hace que cada unidad recurrente capture de forma adaptativa dependencias de diferentes escalas de tiempo. Similarmente a la unidad LSTM, las unidades de activación GRU modulan el flujo de información dentro de la unidad; sin embargo, a diferencia de la unidad LSTM, **no tienen celdas de memoria separadas**.

Con la finalidad de hacer una comparativa entre las redes RNN, LSTM y GRU, consideramos las siguientes redes:

RNN

```
Model: "sequential_20"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_20 (Embedding)     (None, None, 16)          160000
_________________________________________________________________
simple_rnn_4 (SimpleRNN)     (None, None, 16)          528
_________________________________________________________________
simple_rnn_5 (SimpleRNN)     (None, 32)                1568
_________________________________________________________________
dense_27 (Dense)             (None, 64)                2112
_________________________________________________________________
dense_28 (Dense)             (None, 82)                5330
=================================================================
Total params: 169,538
Trainable params: 169,538
Non-trainable params: 0
```

```
RNN
Tokenizer:  10000
Epocas:  5
Batch size:  128
Optimizador:  adam
Loss function:  Binary Crossentropy
Training Loss:  0.1353676161787786
Validation Loss:  0.13734209863410135
Training Accuracy:  0.96040964
Validation Accuracy:  0.9596065
```

LSTM

```
Model: "sequential_19"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_19 (Embedding)     (None, None, 16)          160000
_________________________________________________________________
lstm_27 (LSTM)               (None, None, 16)          2112
_________________________________________________________________
lstm_28 (LSTM)               (None, 32)                6272
_________________________________________________________________
dense_25 (Dense)             (None, 64)                2112
_________________________________________________________________
dense_26 (Dense)             (None, 82)                5330
=================================================================
Total params: 175,826
Trainable params: 175,826
Non-trainable params: 0
```

```
LSTM
Tokenizer:  10000
Epocas:  5
Batch size:  128
Optimizador:  adam
Loss function:  Binary Crossentropy
Training Loss:  0.13532206459079454
Validation Loss:  0.13733660565666442
Training Accuracy:  0.96040976
Validation Accuracy:  0.9596064
```

GRU

```
Model: "sequential_18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_18 (Embedding)     (None, None, 16)          160000
_________________________________________________________________
gru (GRU)                    (None, None, 16)          1584
_________________________________________________________________
gru_1 (GRU)                  (None, 32)                4704
_________________________________________________________________
dense_23 (Dense)             (None, 64)                2112
_________________________________________________________________
dense_24 (Dense)             (None, 82)                5330
=================================================================
Total params: 173,730
Trainable params: 173,730
Non-trainable params: 0
```

```
GRU
Tokenizer:  10000
Epocas:  5
Batch size:  128
Optimizador:  adam
Loss function:  Binary Crossentropy
Training Loss:  0.135260912757463 08
Validation Loss:  0.13728657231092756
Training Accuracy:  0.9604099
Validation Accuracy:  0.9596065
```

Tabla comparativa tiempos

Red	Número de parametros	Tiempo promedio por epoca (segundos)
RNN	169,538	14.6
LSTM	175,826	68.8
GRU	173,730	55.8

Podemos observar en la tabla de arriba que respecto al número de parámetros a entrenar la LSTM es la que tiene el mayor número mientras que la RNN la menor. Por otro lado, considerando el tiempo de ejecución la RNN es la que tiene el menor tiempo mientras que LSTM el mayor. En cuanto al desempeño, podemos ver que para este experimento con los parámetros considerados fue muy similar.

Referencias

- Chung, Junyoung Gulcehre, Caglar Cho, KyungHyun Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.

- Brownlee J.(2017). What Are Word Embeddings for Text? Deep Learning for Natural Language Processing. Consultado el 19 de marzo de 2019 en https://machinelearningmastery.com/what-are-word-embeddings/