TSCF: Aplicacion de redes neuronales recurrentes a slicing en redes WIFI

Federico Grunwald
Marcos Toscano

8 de abril de 2019

Índice

1. Introducción

En este proyecto se investiga la utilización de herramientas de Aprendizaje Automático Profundo en el ámbito de redes Wifi.

En primer lugar se describe el problema que parte de *Slicing in WiFi Networks Through Airtime-based Resource Allocation*.

Luego se presentan herramientas para intentar solucionarlo, se explica el modelo utilizado, incluyendo sus restricciones y decisiones de diseño.

Finalmente se realiza un análisis de los resultados obtenidos y se concluye si el modelo elegido fue satisfactorio o no.

Los datos y código del proyecto de encuentran en el repositorio Git [13] del proyecto.

2. El Problema

El contexto del problema se encuentra en el trabajo *Slicing in WiFi Networks Through Airtime-based Resource Allocation* [1].

Dicho trabajo propone dividir el airtime en distintas slices de forma justa, dada una red WiFi expuesta por un único AP. Cada slice cuenta con un porcentaje dado de dicho airtime. Cuando se solicita el ingreso de una nueva slice, se suma su porcentaje al los porcentajes utilizados por las slices actuales, y sólo si dicha suma es menor o igual a 100 % se acepta la nueva slice.

Sin embargo, se supone un ambiente ideal sin tener en cuenta el ruido del entorno, lo cual trae problemas de precisión cuando se aplica en la practica.

Antes de definir el problema, se mostrara un ejemplo del mismo.

Dado un AP con dos slices utilizando un 30 % y 40 % del airtime respectivamente. En un momento dado, una tercera slice quiere ingresar, solicitando un 30 % del airtime. Asumamos que se acepta dicha slice ya que 30 + 40 + 30 no es mayor a 100. En el ejemplo descrito, el airtime reservado por las slices seria del 100 %. En dicho caso, para satisfacer los requerimientos de las slices, el AP debe utilizar el 100 % del airtime. En el mundo real, hay ruido en el medio, por lo que es improbable que un AP pueda utilizar todo el airtime durante un tiempo prolongado. Se llega entonces, a que la decisión de aceptar la nueva slice, fue incorrecta, ya que por ella, el AP no va a ser capaz de cumplir los requerimientos de las slices.

El problema planteado es análogo a responder a la pregunta: **Dado el historial de estados del AP, ¿se debe aceptar la solicitud de ingreso de una nueva slice para la utilización de un X % de airtime por un tiempo Y?**.

Dicha pregunta no es trivial ya que se necesita predecir como se va a comportar el medio durante el tiempo Y.

3. Investigación

En esta sección se analizan distintos frameworks y modelos de aprendizaje profundo para la resolución del problema.

3.1. Framework

En primer lugar, se analizan los frameworks de aprendizaje profundo disponibles. Dado que los tutores solicitaron que la implementación utilizara TensorFlow, la investigación del problema partió estudiando el mismo.

3.1.1. Tensorflow

Tensorflow [3] es uno de los framework de aprendizaje profundo mas utilizados en la industria, algunas características del mismo:

- Respaldado por Google
- Mejor comunidad
- Mas usado globalmente (estrellas en Github)
- Favorito en la industria (listo para ser utilizado en producción)
- Aprovecha correctamente los diferentes recursos, CPU, GPUs, etc (portable en este sentido).
- **Static computational graph**
- Dificultad en debugging, debido al Static computational graph que no permite insertar break points una vez se comenzó en el entrenamiento del modelo.
- Eager execution: Nueva funcionalidad para tener capacidades parecidas a **Dynamic-Graph**

3.1.1.1 Static computational graph

Tensorflow está basado en la idea de grafos estáticos. Esto quiere decir que se tiene que definir, modelar y crear todas las estructuras internas antes de poder probar el modelo. Gracias a ello Tensorflow es estático al crear y probar los modelos.

Por esto, por ejemplo, se debe saber el tamaño de las inputs del modelo de antemano. Por lo que trabajar con inputs como strings (que tienen largo arbitrario) no es muy directo.

3.1.2. PyTorch

Otro software altamente conocido en el ámbito de Machine Learning, principalmente en el área de investigación es PyTorch [4], algunas características del mismo son:

- Respaldado por Facebook.
- Cada vez se utiliza más en papers científicos.
- Muy útil para prototipar rápidamente.
- Principal competencia de Tensorflow

- **Dynamic Graph**
- Redes modulares, se pueden debuggear separadamente las capas.
- Tiene configuraciones que Tensorflow no tiene, se hace mucho énfasis en la flexibilidad.

3.1.2.1 Dynamic graph

PyTorch está basado en la idea de grafos dinámicos, esto quiere decir que no es necesario definir todo el modelo, incluyendo sus inputs antes de la etapa de entrenamiento. Gracias a esto PyTorch es dinámico al crear/entrenar los modelos.

Otras ventajas de este framework es que el debugging se simplifica y el flujo de control es más flexible (p/ej es posible utilizar capas dinámicas). Como desventaja se tiene que es más lento (comparado con grafos estáticos).

3.1.3. Keras

Keras [5] es una API de alto nivel, creada para abstraer problemas y complejidades que se encuentran al usar softwares complejos como los dos descritos anteriormente. Algunas características que se destacan son:

- Respaldado por Google.
- API de Alto Nivel.
- Backends: TensorFlow, Theano o CNTK.
- Muy directo y simple, ideal para nuevos en el área.
- Minimalista, modular y extensible
- Se van creando (y enpilando) las capas de la red on the go.
- No son necesarios conocimientos amplios en matemática.

3.1.4. Gluon

Gluon [6] es otra API similar a Keras que se ha puesto de moda en los últimos años. Algunas características son:

- Respaldado por Amazon (AWS) y Microsoft (raro, son competencia)
- API de alto nivel,
- Backends: MXNet y pronto Microsoft's CNTK.
- Promete dinamismo en las redes (como PyTorch).
- Competencia directa de Keras.

3.1.5. Decisiones tomadas respecto al software

Se analizaron dichos software entre otros y se concluyó que se utilizaría Keras con TensorFlow de backend. Lo ideal seria utilizar TensorFlow, pero dado el alcance de la materia y del tiempo que consume la misma, Keras pareció la opción mas viable aprovechando que su curva de aprendizaje es considerablemente menor a utilizar TensorFlow directamente.

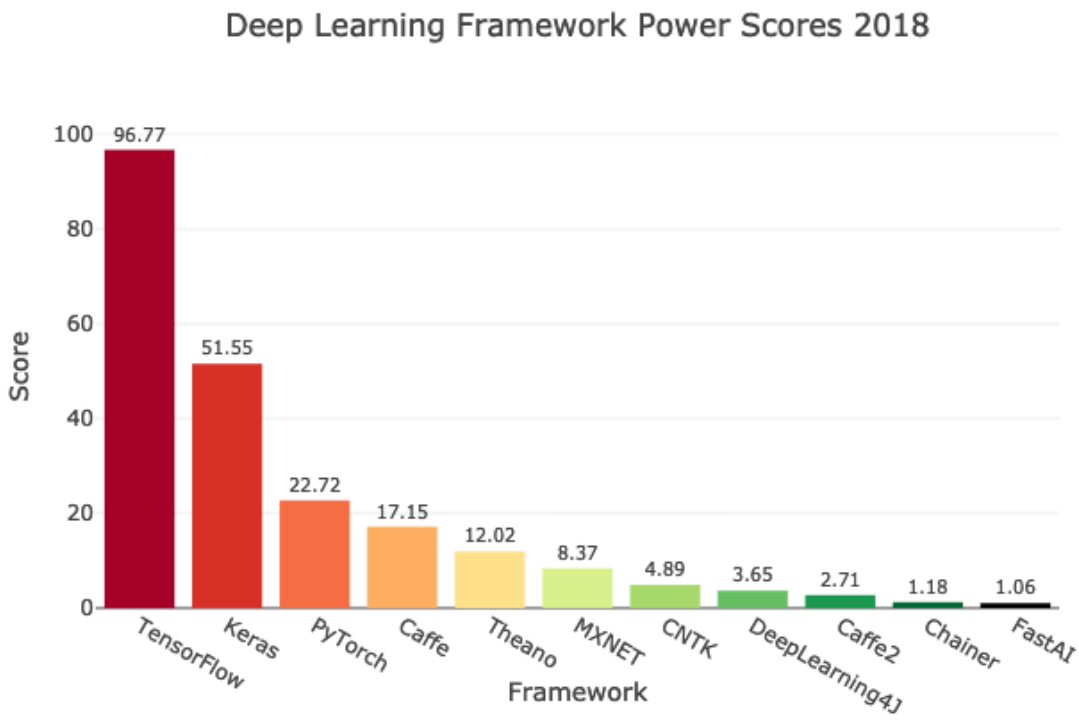

Figura 1: Puntaje de distintos algoritmos [2]

3.2. Modelo

Para atacar el problema planteado se analizaron distintos tipos de algoritmos y se concluyó que los dos mejores para el problema a resolver son:

- Redes neuronales recurrentes, por ejemplo Long short-term memory (LSTM) [7]
- Aprendizaje por Refuerzos

Ambos algoritmos se toman en cuenta ya que incorporan el concepto de memoria, que es necesario para un problema como el que se plantea.

El concepto de memoria es necesario ya que para decidir si una slice debería o no ser aceptada, se debe tener en cuenta el estado actual del AP, así como los estados anteriores del mismo.

Como no hay antecedentes a este tipo de problema, no hay preferencia por un algoritmo en particular. Sin embargo, luego de investigar y discutir con colegas, se decide por utilizar el modelo LSTM ya que en la practica es un modelo mas estable.

3.2.1. Descripción del modelo

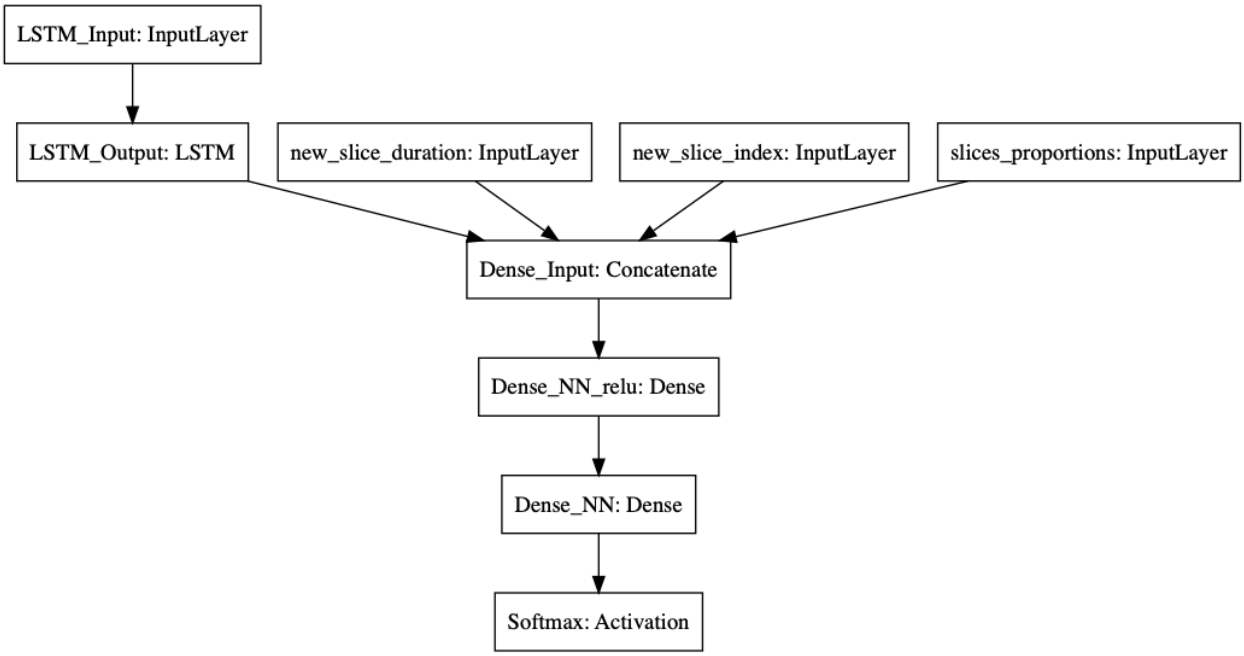

Figura 2: Descripción del modelo

Como se dijo anteriormente se opto por utilizar una red LSTM para resolver el problema.

La misma tiene como entrada una historia, que a grandes rasgos es una lista de datos, que representan lo que sucede temporalmente.

Se notó que hay varios datos que son en común en toda historia, por lo que no tenia sentido agregar dichos datos fijos a todos los elementos de la historia.

El enfoque que se decidió tomar entonces, fue que la LSTM se encargase solamente de predecir como se comportaría el medio y utilizando dicha predicción sumados a los otros datos de entrada, conectarlos a una red neuronal completamente conectada que se encargaría de predecir la salida final del modelo.

Entonces, en primer instancia se utiliza la red LSTM.

Se eligió caracterizar al medio por 8 valores, que son el estado del AP en un momento dado (TX, RX, BUSY e IDLE), el airtime consumido por cada slice y el promedio ponderado de su uso, que intenta describir la diferencia entre el uso actual de las slices y el uso esperado de las slices.

Se decidió tomar 100 datos anteriores a la entrada de la nueva slice para tener un historial del estado del medio.

Se eligieron 100 datos ya que viendo el tamaño de las historias generadas, dicho número pareció razonable.

Dicho hiperparámetro puede ser cambiado, y al hacerlo es necesario preprocesar todas las historias nuevamente.

Por lo tanto el tamaño de la entrada de la LSTM es de $100 * 8$, dichos datos corresponden a $LSTM\_Input$ del modelo.

La salida de la LSTM ($LSTM\_Output$ del modelo) es un vector y su dimensión es un hiperparámetro. Dicho vector intenta caracterizar como se comportara el medio en el futuro cercano.

Luego se conecta la red neuronal densa (completamente conectada) de dos capas, cuya entrada es la concatenación de:

- La duración de la nueva slice ($new\_slice\_duration$ en el modelo).
- El slot de la nueva slice, o sea cual de las 3 slices es la que ingresa (new_slice_index en el modelo).
- Las proporciones de las slices ($slices\_proportion$ del modelo).
- La salida de la LSTM ($LSTM\_Output$ del modelo).

Dicha red es la encargada de predecir si el ingreso de la nueva slice es una decisión correcta o no.

Dicha decisión se toma tomando en cuenta como se comportara el medio (salida LSTM), y la información de la nueva slice (los datos restantes).

Finalmente, la salida de la red neuronal densa pasa por una capa de softmax que arroja 2 salidas con valores entre 0 y 1, los cuales intentan responder la pregunta del problema, o sea si una slice debería ser aceptada o no.

3.3. Datos de entrada

El mayor desafío del problema son los datos a utilizar. En un principio no se sabia de dónde se conseguirían, ni que forma tendrían, ni como debían ser procesados. La primer respuesta a todas estas preguntas fue, de donde vendrían.

Los datos en esta instancia fueron sacados de un simulador por los tutores de la materia.

Sobre los datos crudos fue necesario hacer un preprocesado complejo, para que sean compatibles con el modelo.

3.3.1. Formato de datos crudos

Los datos recibidos están separados en distintos escenarios del simulador. Un ejemplo de un escenario seria:

- En minuto 0, ingresa slice 1 que solicita el 20 % de los recursos durante 24 horas.
- En minuto 0, ingresa 1 cliente para la slice 1.
- En minuto 4 (240), ingresa 1 cliente para la slice 1.
- En minuto 6 (360), ingresan 2 clientes para la slice 1.

- En minuto 9 (540) ingresa slice 2 que solicita el 20 % de los recursos durante 12 horas.
- En minuto 9 (540) ingresan 4 clientes para la slice 2.
- En minuto 11 (660) ingresa slice 3 que solicita el 60 % de los recursos durante 13 horas.
- En minuto 11 (660) ingresan 2 clientes para la slice 3.
- En minuto 15 (900) ingresan 2 clientes para la slice 3.

Recursos totalmente utilizados hasta el minuto 21 (1260) que termina la slice 2.

Los datos crudos cuentan con los siguientes elementos: A nivel de AP:

- TX
- RX
- BUSY
- IDLE

A nivel de Slice:

- Airtime consumido por cada slice

Cada elemento corresponde a una unidad de tiempo (el cual puede interpretarse como segundos, minutos, horas).

3.3.2. Estructura de datos crudos

Los datos tiene una estructura que debe respetarse al agregar nuevos escenarios.

Los mismos se encuentran en una carpeta $slicing-learning-experiments$ del repositorio. Cada escenario se encuentra en su carpeta, llamada $EscenarioX$ donde X es el número de escenario.

Dentro de dicha carpeta se encuentra un archivo llamado $config.txt$ que contiene la configuración de las slices para dicho escenario.

Un ejemplo de dicho archivo sería:

```
0.2
0.2
0.6
```

Cada linea es el porcentaje de cada slice.

Luego se encuentran dos carpetas mas, llamadas *airtime* y *channel* las cuales contienen información del airtime de las slices y del canal respectivamente.

Dentro de la carpeta *airtime* se utiliza un archivo llamado $airtime-ratio-all-slices.txt$. Dicho archivo no debe contener cabezales y su formato es:

```
692 0.107908 0.132328 0.37784
693 0.13328 0.113464 0.342336
694 0.111492 0.123944 0.374352
695 0.118772 0.12446 0.361524
696 0.121808 0.125832 0.377028
697 0.127584 0.130212 0.385692
698 0.127804 0.111636 0.362496
```

El primer número de cada linea es el timestamp (que es común con otros archivos) los demás, es el uso que hace cada slice del airtime en dicho tiempo.

Dicho archivo se genera utilizando scripts, a partir de la salida del simulador. Dichos scripts se encuentran en la carpeta $/slicing - learning - experiments/scripts\_airtime$.

Dentro de la carpeta *channel* se encuentran 3 archivos: $busy - minstrelHT - 0.plt$, $idle - minstrelHT - 0.plt$, $rx - minstrelHT - 0.plt$, $tx - minstrelHT - 0.plt$. Dichos archivos continen datos de BUSY, IDLE, RX, y TX del AP por unidad de tiempo.

El formato de dichos archivos es así:

```
0 0
1 82.7168
2 72.9206
3 73.2534
4 73.382
5 73.4131
6 73.5011
7 73.4084
8 73.409
```

Dónde el primer número es el timestamp y el segundo es el dato per se. Notar que estos 4 archivos contienen un 0 0 al principio, mientras que $airtime - ratio - all - slices.txt$ no.

Los datos fueron entregados así, se supone que si es el output del simulador.

Los archivos nombrados son los efectivamente se utilizan para generar el dataset.

3.3.3. Clasificación de los datos

Una pregunta interesante que surgió fue: *¿Cómo se sabe si una slice fallo?* Para responder dicha pregunta, primero se definió que una slice no cumple con sus requerimientos si en un momento dado su airtime es menor al requerido por la slice menos un `K=10%`.

Por ejemplo, si una slice tiene un requerimiento del `40%`, con el `K=10%`, la slice falla si su uso de airtime es menor a `36%`.

Respondiendo a la pregunta original, se definió que *Una slice falla sí hay una ráfaga de 100 unidades de tiempo donde la misma no cumple con su airtime requerido*.

Dicha pregunta se respondió estudiando los datos que se tenían en dicho momento, se intento tener en cuenta la calidad de la conexión del usuario. Este dato es utilizado para etiquetar si la decisión de aceptar la entrada de la nueva slice fue correcta o no.

3.3.4. Preprocesado de los datos

Como se dijo arriba, las redes LSTM obtienen como entrada una historia, que dentro tiene varios datos. En este caso se definió que cada historia conceptualmente será la entrada de una nueva slice y esta conformada por:

- Datos fijos:
  - Porcentaje requerido por la nueva slice.
  - Duración de la nueva slice.
  - Configuraciones de las demás slices.
  - Resultado de la historia (etiqueta).
- Datos dinámicos:
  - Datos de la historia, los 100 anteriores a la entrada de la slice:
    - RX
    - TX
    - IDLE
    - BUSY
    - Uso de las Slices
    - Error ponderado

Se decidió tomar dichos datos, ya que los mismos fueron considerados como los que mas podrían caracterizar al medio.

Los datos dinámicos son usados para entrenar la red LSTM, mientras que los datos fijos son utilizados junto a la salida de la red LSTM para llegar al resultado final.

Luego del preprocesado de los datos, se obtienen historias. Cada historia representa la entrada de una slice al sistema, y contiene la siguiente información:

```
{
  "new_slice_duration": 779, // Duración de la nueva slice.
  "new_slice_index": 2, // Indica si la nueva slice es la 1, 2 o 3.
  "slices_proportions": [ // Indica las proporciones de las slices en la historia.
    0.2, // Proporción de la Slice 1
    0.2, // Proporción de la Slice 2
    0.6 // Proporción de la Slice 3
  ],
  "history_elements": [ // Los elementos de la historia
    [
      0.541476, // RX
      0.0, // TX
      0.0, // IDLE
      0.0, // BUSY
      29.3198, // Uso de Slice 1
```

```
      16.8356, // Uso de Slice 2
      53.8616, // Uso de Slice 3
      0.1082952 // Error ponderado
    ],
    ...
  ],
  "result": true // Resultado de la historia
}
```

Este será el formato de entrada para el modelo.

3.3.5. Etapas del preprocesado

Se explicara el preprocesado de los datos, desde los datos crudos (salida del simulador) hasta el formato de entrada del modelo.

Dicho preprocesado tiene 5 etapas. Se explicará cada etapa, las mismas están en el Notebook del proyecto, con los mismos nombres y es recomendable leerlas para el mejor entendimiento de cada una, ya que están comentadas.

Primer etapa:

Esta etapa es llamada *Unificar datos por escenario* se encarga de tomar los distintos archivos nombrados anteriormente, y unificarlos en objetos.

Para ellos se carga cada archivo en un diccionario.

Luego se unen los los distintos archivos que contienen los datos temporales:

- $/airtime/airtime - ratio - all - slices.txt$
- $/channel/busy - minstrelHT - 0.plt$
- $/channel/idle - minstrelHT - 0.plt$
- $/channel/rx - minstrelHT - 0.plt$
- $/channel/tx - minstrelHT - 0.plt$

Cada uno tiene un timestamp, se unen dichos datos utilizando el timestamp como clave, o sea que dado un timestamp X se toma el TX, RX, IDLE, BUSY, y airtimes que correspondan al timestamp X.

Al final de esta etapa se tiene por escenario, los datos nombrados unidos.

Segunda etapa:

Esta etapa es llamada *Etiquetar y formatear datos* y se encarga de etiquetar los datos como se describió en la sección anterior.

Dentro de cada escenario se analizan los datos de cada slice en el correr del tiempo.

Para ello se determina para cada instancia de una slice, el tiempo en el que inicia y el tiempo en el que termina.

Entre dicho intervalo, para cada unidad de tiempo, se analiza si la slice esta cumpliendo su requisito o no, en caso negativo, se le suma 1 al contador que cuenta las veces que falla cada slice.

Si en algún momento dado, dicho contador pasa un umbral de 100 fallas seguidas, se etiqueta dicha slice como fallida, dicha etiqueta representa si fue una buena decisión aceptar dicha slice o no.

Cada intervalo nombrado, se corresponde con una historia.

Como la red LSTM requiere un largo fijo, se toman los 100 datos anteriores a la entrada de una slice.

Una nota importante, es que las slices que inician en el instante 0, no pueden ser contadas, ya que no hay 100 datos anteriores a ellas.

Se recomienda que al crear escenarios nuevos se tenga esto en cuenta, y esperar al menos 100 unidades de tiempo para agregar una slice nueva.

Tercera etapa:

Esta etapa se trata de *Data Augmentation*, esto es, utilizando las historias generadas, generar más.

El Data Augmentation pensado parte de las siguientes premisas:

- Si una historia con una slice de duracion X es positiva, la misma seguira siendo positiva para tiempos $Y|Y <= X$
- Si una historia con una slice de duracion X es negativa, la misma seguirá siendo negativa para tiempos $Y|Y >= X$

Por lo tanto por cada historia se generaban 5 mas, con un tiempo 10 %, 20 %, 30 %, 40 % y 50 % mas grande/chico.

Cuarta y quinta etapa:

Dichas etapas simplemente empacan y desempacan los datos para poder separarlos en un set de entrenamiento y otro de prueba.

Se separa el dataset en 33 % de prueba y un 67 % de entrenamiento.

La función utilizada para separar dichos datos admite solamente una lista con datos simples, mientras que los datos obtenidos son varias listas, no una.

Por lo tanto se agrupan los datos tal que se obtiene una sola lista.

Luego de agrupar dichos datos y separar en los distintos sets, se desagrupan nuevamente.

3.4. Hiperparámetros

Luego de definir el modelo se estudiaron los posibles hiperparámetros que podrían aumentar el rendimiento del modelo.

Ya que el modelo planteado utilizada multi-entradas (o sea que contiene varios elementos y algunos son listas) no servía cualquier software de tuneo de hiperparámetros.

Investigando se logró conseguir un software que era compatible con el modelo, era simple y no requería grandes cambios de código.

Se decidió entonces, utilizar la librería Talos[8], la cuál dada una lista hiperparámetros, entrena el modelo con todas las combinaciones posible, retornado la combinación que muestre un mejor resultado.

Talos cuenta con una manera sencilla de exportar su resultado (el mejor modelo) en un formato para utilizable en producción, lo cual puede ser útil en el proyecto.

Un problema con utilizar este tipo de software es que entrenan el modelo miles de veces, cambiando los hiperparámetros, en todas sus combinaciones posibles, por lo que el tiempo que tomaba en correr era exponencial a la cantidad de hiperparámetros.

Sumado a que las pruebas se hacían en notebooks comunes (sin GPUs) se decidió minimizar la cantidad de hiperparámetros a tener en cuenta.

Los hiperparámetros elegidos son los mas comunes en este tipo de modelos, se discutió con colegas las mejores opciones para dicha elección.

La implementación de LSTM utilizada (la de Keras) no cuenta muchos hiperparámetros, lo cual puede ser una gran desventaja.

A continuación se listaran los hiperparámetros que se intentan optimizar.

```
{
  "lr": [0.1, 0.8, 5],
  "number_of_hidden_neurons":[2, 4, 8],
  "epochs": [100],
  "dropout": [0, 0.2, 0.5],
  "lstm_out_dimension": [1, 4, 8, 16],
  "lstm_dropout": [0, 0.25, 0.5],
  "optimizer": [Adam, Nadam],
  "losses": ["logcosh", "binary_crossentropy"],
  "activation":["relu"]
}
```

- lr - Learning Rate, hiperparámetro que determina en qué medida la información recién adquirida anula la información antigua.
- number_of_hidden_neurons - Cantidad de neuronas de la capa oculta en la red neuronal densa.
- epochs - Cantidad de iteraciones con las que se entrena el modelo.
- dropout - Porcentaje de dropout[9] de la red neuronal densa.
- lstm_out_dimension - Dimensión de la salida de la red LSTM.
- lstm_dropout - Porcentaje de dropout[9] de la red LSTM.
- optimizer - Que Keras Optimizer[10] utilizar.
- losses - Que funcion de perdida[11] utilizar para entrenar el modelo.
- activation - Que función de activación[12] utilizar en la red neuronal densa.

3.4.1. Talos en la práctica

Talos como input tiene simplemente una función y un diccionario con los hiperparámetros a tunear (exactamente como se muestra arriba).

Dicha función (*input_model*) tiene como entrada el set de entrenamiento y el diccionario de hiperparámetros descrito anteriormente. En dicha función se define el modelo de Keras con la diferencia de que en cada lugar donde iría un hiperparámetro, se pone al diccionario de hiperparámetros.

Por ejemplo, al definir el modelo en un principio, la siguiente linea se usaba para entrenar el modelo:

```
history = model.fit(
  x_train,
  y_train,
  epochs=100
)
```

Donde se ve que los *epochs* son 100. Al utilizar Talos, esta linea se transformaría en:

```
history = model.fit(
  x_train,
  y_train,
  epochs=params['epochs']
)
```

Donde *params* es el diccionario de hiperparámetros.

Con la configuración descrita, se entrenaron mas de 5000 modelos, lo que tomó aproximadamente 20 horas.

4. Resultados

Todas las pruebas en esta sección se hacen con las historias generadas, que son 22, ordenadas de forma aleatoria divididas en 14 datos de entrenamiento y 8 de prueba.

Los resultados, incluso antes del tuneo de hiperparámetros fueron pobres. Se probaron varios miles de modelos distintos (con hiperparámetros diferentes) y los resultados fueron similares.

- La precisión rara vez superaba el 80 %.
- En varias de las pruebas el modelo terminaba decidiendo responder siempre que no.
- La precisión y perdida muchas veces oscilaba.

En un principio de intento hacer Data Augmentation. Sin embargo se descarto dicha opción, ya que no ayudó en absoluto, y los tiempos de entrenamiento aumentaban.

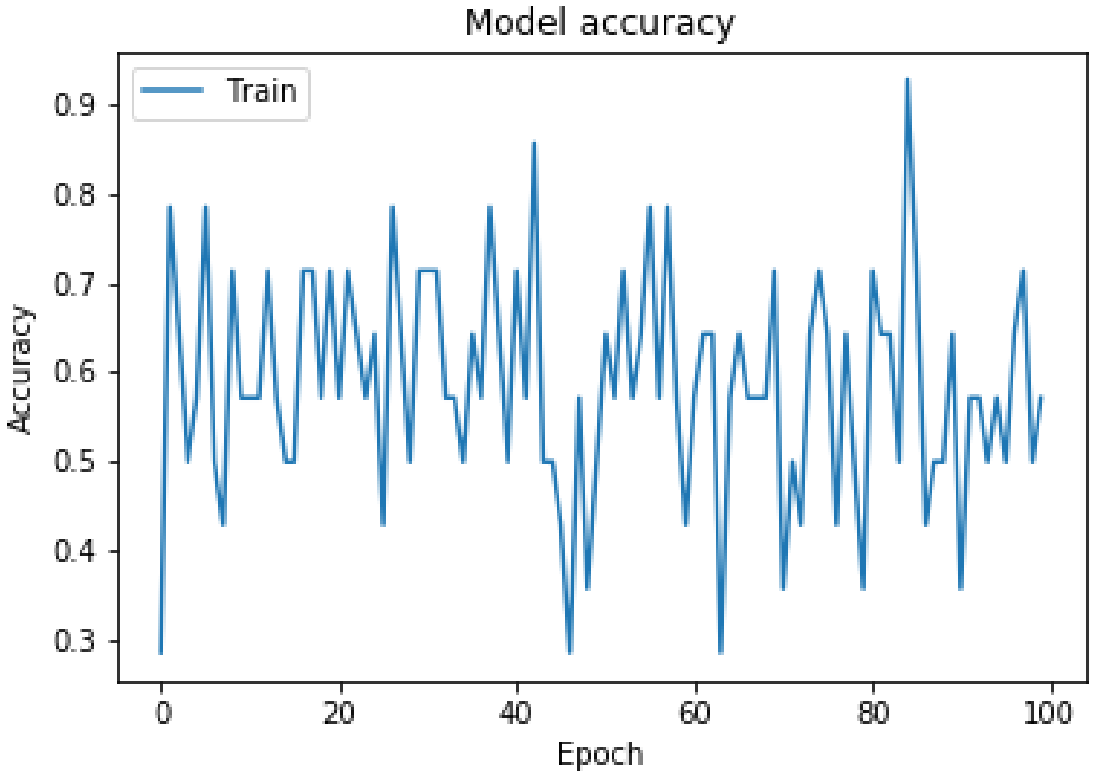

Figura 3: Precisión del modelo

4.1. Modelo Final

Los hiperparámetros elegidos por Talos fueron:

```
{
  "lr": 0.1,
  "number_of_hidden_neurons": 8,
  "epochs": 100,
  "dropout": 0.5,
  "lstm_out_dimension": 8,
  "lstm_dropout": 0,
  "optimizer": Adam,
  "losses": "binary_crossentropy",
  "activation": "relu"
}
```

Finalmente se probo el modelo final con los 8 datos de prueba, los cuales no fueron tenidos en cuenta en ninguna etapa de entrenamiento, de dichos elementos el modelo predijo correctamente solo 5 datos

Los resultados obtenidos son los siguientes:

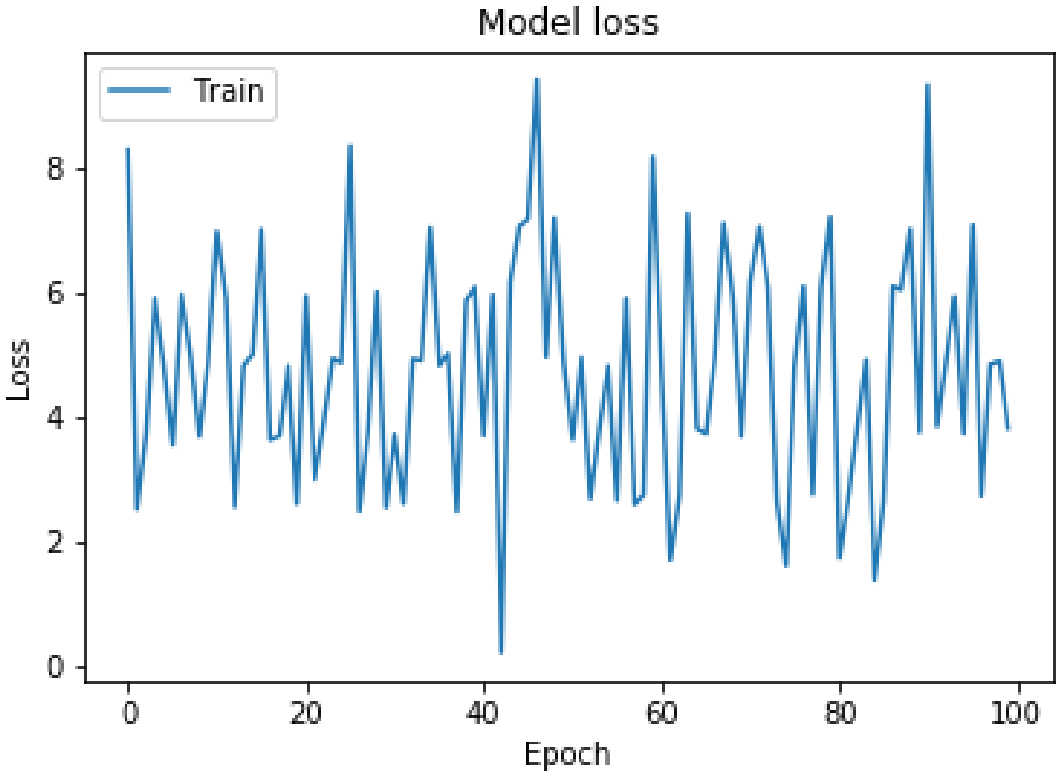

Figura 4: Pérdida del modelo

```
Algoritmo Original: SI
Modelo: SI
Resultado: NO

Algoritmo Original: SI
Modelo: NO
Resultado: SI

Algoritmo Original: SI
Modelo: SI
Resultado: SI

Algoritmo Original: SI
Modelo: NO
Resultado: NO

Algoritmo Original: SI
Modelo: NO
Resultado: NO
```

```
Algoritmo Original: SI
Modelo: NO
Resultado: NO

Algoritmo Original: SI
Modelo: NO
Resultado: SI

Algoritmo Original: SI
Modelo:  NO
Resultado: NO
```

4.1.1. Probar el modelo final

Para probar el modelo final, se creó un script llamado *predict_history.py* que toma como parámetro un archivo JSON que como contenido tenga una o mas historias, utilizando el formato descrito en el preprocesado de los datos.

El archivo se ejecuta de la siguiente forma:

python predict_history.py archivo_con_historias.json

La salida son duplas de números, por ejemplo:

0.75 0.20

Esto se interpreta en que el modelo predice con un 75 % de probabilidad, que aceptar la slice de la historia es una buena decisión, y un 20 % de que no.

5. Conclusiones

Los resultados no fueron los esperados.

Se plantean dos posibles explicaciones de cual es el problema.

En primer lugar, se cuenta con una baja cantidad de datos (solamente 22 historias); con mas datos y volviendo a optimizar los hiperparámetros, los resultados podrían ser totalmente distintos.

Por otra parte, los datos están desbalanceados, la mayoría de los datos son negativos, 15 negativos contra solamente 7 positivos. Un aspecto que hizo que los datos sean desbalanceados es el hecho de que en todas las historias, la suma de las 3 slices da 100 %.

Esto es un peor caso ya que como se dijo anteriormente en estos casos es altamente probable que la ultima slice ingresada sea errónea.

Si se agregaran datos en los cuales la suma de las slices sea menos a 100 % probablemente el modelo logre aprender mejor.

Sobre los resultados del modelo final, se concluye:

- 2 slices son rechazadas cuando deberían haber sido aceptadas.
- 2 slices son correctamente rechazadas, cuando el algoritmo original las hubiera aceptado.

- 1 slice es aceptada correctamente, por el modelo y el algoritmo original.
- 1 slice es incorrectamente aceptada por el modelo; el algoritmo original hubiera aceptado dicha slice igualmente.

Comparando entonces los resultados con el algoritmo original, de los 8 datos, solamente las 2 slices rechazadas incorrectamente hubieran sido peores que el algoritmo original, ya que se estarían rechazando 2 slices que posiblemente hubieran funcionado bien.

El error restante (la otra slice incorrectamente aceptada) no es estrictamente peor que el algoritmo original, ya que el mismo la hubiera aceptado de todas formas.

Por lo que si bien el error real del modelo en este caso fue un 62.5 %, en el 75 % de los casos, el modelo es igual o mejor que el algoritmo original.

6. Trabajo futuro

Se listan aquí algunas lineas de trabajo futuro.

- Conseguir una mayor cantidad de datos.
- Conseguir datos reales, o sea, no simulados.
- Investigar mas a fondo el tuneo de hiperparámetros.
- Hacer cambios en el modelo para permitir situaciones con mas de 3 slices.
- Cambiar Keras por TensorFlow 2.0, que implementa una API similar a la de Keras.
- Utilizar una implementación de LSTM que tenga mas hiperparámetros.
- Intentar modelar el problema descrito utilizando Aprendizaje por Refuerzos.
- Analizar los datos del dataset, y remover redundancias.

Referencias

[1] Richart, M., Baliosian, J., Serrat, J. et al. J Netw Syst Manage (2018). https://doi.org/10.1007/s10922-018-9484-x

[2] Deep Learning Framework Power Scores 2018, `https://www.kaggle.com/discdiver/deep-learning-framework-power-scores-2018`

[3] Tensorflow Official Guide, `https://www.tensorflow.org/guide`

[4] PyTorch Official Tutorials, `https://pytorch.org/tutorials/`

[5] Keras Official (Tensorflow) Guide,
https://www.tensorflow.org/guide/keras

[6] Gluon Official Documentation,
https://gluon.mxnet.io/

[7] Illustrated Guide to LSTM,
https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

[8] Talos Documentation,
https://github.com/autonomio/talos

[9] Dropout: A Simple Way to Prevent Neural Networks from Overfitting,
http://jmlr.org/papers/v15/srivastava14a.html

[10] Keras Optimizers,
https://keras.io/optimizers/

[11] Loss function,
https://en.wikipedia.org/wiki/Loss_function

[12] Activation function,
https://en.wikipedia.org/wiki/Activation_function

[13] Project Git Repo,
https://gitlab.fing.edu.uy/marcos.toscano/tscf