Detección de Botnets usando Redes Neuronales

Datos del estudiante

- **Nombre:** Sergio
- **Apellidos:** Roselló Morell
- **DNI:** 53632974X
- **email:** Sergio-resello@hotmail.con

Información sobre el entorno

- **Sistema Operativo:** Arch Linux
- **Entorno de escritorio:** dwm
- **Versión de Python:** Python 3.8.2
- **Editor de texto:** NeoVim
- **Generación del documento:** Escrito en Markdown, compilado a LaTeX con `pandoc`

  ```
  nnoremap <leader>e :! pandoc % -f markdown -t latex -s
  -o %:r.pdf<cr>
  ```

Resumen

En el trabajo de investigación a continuación, se va a revisar el uso de redes neuronales para catalogar y detectar Botnets en la red. Los pasos que se realizan engloban de forma general cualquier problema de Machine Learning, en el que se obtiene el dataset, se trata, para que contenga los valores deseados, se entrena la red neuronal, se genera el modelo y se revisa, para luego refinar el resultado.

Indice

Planteamiento

Descripción y planteamiento del problema

El problema que se plantea en este análisis es identificar tráfico de red malicioso en una red a tiempo real.

La solución debe poder analizar a tiempo real los paquetes generados por la red y decidir si cada paquete individual es o no un paquete proveniente de un malware.

Este problema se adapta muy bien a una solución relacionada con Inteligencia Artificial. Mas en concreto, a modelos como las redes neuronales.

El principal inconveniente a la hora de solucionar un problema de selección en tiempo real con modelos tradicionales, aparece cuando se pretende analizar el tráfico y compararlo con varios ejemplos de malware. Ya sea comparando directamente pequeños atributos, como el paquete de red entero, al final, estamos comparando con características que ya conocemos e identificamos como maliciosas. Esto quiere decir que estamos reaccionando al problema, no tomando medidas pro-activas al problema en cuestión. La finalidad del problema es ser capaces de detectar malware, aunque no se haya detectado anteriormente el tipo específico de malware siendo analizado.

Solucionar el problema en cuestión con métodos tradicionales, como bien puede ser comprobaciones secuenciales de características del paquete de red a analizar incrementa rápidamente la complejidad del algoritmo. Además, cada vez que se detecten nuevos casos, se debe integrar la comprobación al programa. Esto hace que sea imposible mantenerlo actualizado.

Existen otras técnicas, no tan primitivas, como por ejemplo comprobación de hashes, tanto completos, como parciales del paquete de red y sus características. Si podemos identificar las características comunes en los paquetes de red enviados entre el malware y el servidor C&C, podemos cifrar estos datos en hashes. Una vez tenemos suficiente información sobre los hashes malignos, los podemos comparar contra los nuestros propios y determinar si tenemos o no paquetes malware en nuestra red.

Como extensión a la revisión de hashes, se podría hacer una plataforma a la que se suban todos los hashes de paquetes de red que han sido identificados como `Botnet`. Nuestro programa, revisa los hashes obtenidos en nuestra red contra los definidos como maliciosos en la plataforma.

Esta es la base de los Antivirus, aunque plataformas como virustotal llevan los esfuerzos de detección a la comunidad, en vez de a las empresas individuales.

De cualquier forma, este enfoque se sigue pudiendo adaptar a los modelos de redes neuronales, en los que cada X tiempo se actualice el modelo a el más nuevo, generado por muestras de tráfico benigno y maligno.

En resumen, los modelos basados en redes neuronales ofrecen una mejora sustancial en la trata de datos masiva y discriminación de características basada en ellos.

Descripción de los datos a utilizar

Para desarrollar un modelo de detección de paquetes maliciosos, lo más importante es la calidad de los datos iniciales que tenemos. Si generamos el modelo con datos buenos, el modelo puede inferir, en muchas ocasiones el tráfico de red maligno.

Es muy importante tener unos datos tanto específicos, como generales, con distintas muestras y combinaciones de paquetes de red malignos, ya que estos son la parte más crítica del proyecto.

Como se ha mencionado superficialmente en la sección anterior, un modelo de detección basado en una red neuronal esta preparado para operar en un entorno de producción, en el que los paquetes de red que va a revisar no son exactamente iguales que con los que ha entrenado, de esta forma, entrenando con un conjunto de datos lo suficientemente rico, podemos inferir la clase del paquete de red.

En el caso del entrenamiento del modelo, vamos a seleccionar un dataset realista, que contenga tanto tráfico de red benigno como maligno y varios ejemplos de cada tipo.

Selección de Dataset de entrenamiento Se ha usado la pagina web de www.secrepo.com para buscar un dataset que contenga las propiedades deseadas para el estudio.

Las cualidades que se necesitan en el dataset son:

- Tráfico de malware hacia servidores C&C
- Tráfico de aplicaciones no maliciosas
- Cantidad de información (Necesario para poder inferir comportamientos y generar modelos de datos)
- Calidad de información (Ejemplos de tráfico claro)

Siguiendo los requisitos demarcados anteriormente, se han encontrado varios datasets, entre estos, se va a usar `ISOT HTTP Botnet Dataset`, desarrollado por Alenazi A y compañeros para una charla con titulo: “Intelligent, Secure, and Dependable Systems in Distributed and Cloud Environments”.

Este dataset ha sido generado por la universidad de Victoria en el año 2017 y consiste en nueve capturas de tráfico malicioso y 19 capturas de tráfico de aplicaciones no maliciosas, como por ejemplo Dropbox o Avast.

Cada registro se puede trazar directamente a una IP que contiene únicamente el programa a analizar, ya sea malware o no. Esto quiere decir que el dataset es claro, de forma que se puede analizar de forma manual y llegar a una serie de

conclusiones desde el primer momento. Esto nos permite deducir los posibles resultados del análisis con el algoritmo de aprendizaje automático.

Es importante que tanto los paquetes maliciosos como los corrientes se hayan capturado al mismo tiempo ya que es una de las formas que tenemos para generar un fragmento de entrenamiento y test verídico.

Una de las ventajas de este dataset frente a otros que también cumplían los requisitos es que los datos vienen en un formato `.pcap`. Este detalle permite al investigador tomar el control de la información que se va a añadir al `.csv` para proporcionar al programa de generación del modelo.

Datos sobre los que opera el modelo en producción Una vez este el modelo terminado, va a ser puesto en un punto estratégico de la red, en el que tiene visibilidad de los paquetes entrantes y salientes de la misma. Un sitio en el que podría estar es en el router o switch de la red, actuando de firewall.

Cuando esté desplegado, este sistema revisa cada paquete que entra o sale de la red y avisa (En caso de IDS) o ejecuta medidas preventivas (En caso de IPS) según el tipo de paquete que detecte.

En principio, si el modelo ha sido entrenado correctamente, no hace falta volver a entrenarlo con más datos, pero si lo ponemos y vemos que no detecta correctamente el tipo de paquete que analiza, puede que tengamos un problema de especificidad de datos. El dataset que hemos seleccionado para entrenar el modelo es bastante específico, tiene datos concretos de aplicaciones concretas, como Dropbox, pero no de tráfico realista de red, como usuarios buscando cosas en Google o otras aplicaciones diversas. En caso de que el modelo no detecte correctamente los paquetes maliciosos, la solución es obtener una muestra de nuestra propia red y analizar su tráfico, junto con muestras de botnets.

Posibles algoritmos de aprendizaje automático a usar

En la asignatura cursada, se han realizado estudios sobre algoritmos de 'clustering', Clasificación, Probabilidad y 'Deep Learning'

Teniendo en cuenta los algoritmos que se han aprendido durante el curso, se decide ahondar más en los recursos sobre `Deep Learning`.

Los motivos por esta selección, entre otros son:

- Modelo adaptado al problema
- Mas sencillo que el problema con el enfoque tradicional (Secuencial)
- Manejo de datos eficiente
- Versatilidad

Los modelos secuenciales, aunque igual de eficientes y rápidos de implementar que los modelos neuronales con datasets sencillos y pequeños, empiezan a aumentar en complejidad a medida que los modelos aumentan en tamaño y complejidad. Los modelos de redes neuronales no tienen esa desventaja, ya que una vez se ha

realizado la primera implementación, modificarla para que acomode más datos es trivial. Mas aun con los frameworks de los que disponemos, como Keras o TensorFlow.

Breve comparación de los modelos y paradigmas Más allá de las diversas implementaciones, existen varios paradigmas donde los más usados/reconocidos son:

- Supervisado
- No Supervisado
- Aprendizaje por Refuerzo
- Auto-aprendizaje

Supervisado La meta de este paradigma es generar la misma salida que lo esperado. Son muy eficaces para detectar patrones (Justo nuestro dominio) y regresión

Es el paradigma que vamos a usar en esta práctica. Tenemos la información necesaria para esto, ya que tenemos clasificados los paquetes de red en `Botnet` y común.

No Supervisado A este modelo se le pasan los datos de entrada, además de la función de coste, que es la que determina si el modelo esta progresando adecuadamente. Algunas áreas en las que este paradigma es relevante son: Probabilidad, Clustering o Distribución estadística

Aprendizaje por Refuerzo La meta es generar un modelo que minimice el coste, dada una entrada de datos. Por ejemplo, este modelo es bueno en juegos, en los que el modelo recibe información y debe generar el siguiente mejor movimiento.

Auto-aprendizaje Es un sistema con una entrada (Situación) y una salida (Acción). Se adapta a la salida previa, para generar la nueva salida. Un ejemplo de estos algoritmos son los algoritmos genéticos.

Dentro del campo de las redes neuronales, existen varias implementaciones, cada una especifica para su propio dominio y especialidad. Entre estas, las más reconocidas son:

- Redes neuronales Convolucionales
  - Buenas en análisis de imágenes y datos bidimensionales
- Long Short-Term Memory
  - Eliminan el problema del gradiente
  - Buenas con análisis de vocabulario

La que se va a implementar en esta práctica no es tan sofisticada como las descritas anteriormente, en parte porque no es necesario, y en parte por mi experiencia.

Por qué Redes neuronales Se podría usar clustering para solventar el problema, pero tenemos demasiados datos, que desaprovecharíamos si usáramos un algoritmo de clustering. Digo esto porque Clustering es un algoritmo de aprendizaje no supervisado, mientras que nosotros tenemos la capacidad de implementar un algoritmo supervisado, porque sabemos a qué clase `Botnet` pertenece cada tupla. Además, clustering agrupa datos basándose en una combinación de características que componen un paquete en este caso, pero como hemos dicho anteriormente, ya sabemos como debemos agrupar los datos.

Entre los distintos tipos de redes neuronales existentes, se ha optado por entrenar el modelo con capas Dense, que son la forma más básica de generar una red neuronal.

Gestión de datos (Entrenamiento, modelado, normalización, categorización) Hay una serie de cambios que son necesarios hacer cuando se adapta un dataset de datos crudos a un dataset valido para ser la entrada de un modelo.

Los pasos que se van a tener que seguir son:

- Seleccionar *features* con las que nos quedamos
- Convertir datos a `csv`
- Sanado de el dataset
- Determinar datos categóricos y continuos
- Codificar datos categóricos y continuos
- Dividir el dataset en datos de prueba y entrenamiento

Elección de 'features' Una de las decisiones más importantes cuando seleccionamos un dataset es saber que los datos están disponibles de la forma más pura posible. Esto hace que sea más complicado trabajar con ellos desde un inicio, debido a que se tienen que convertir y mutar para que sirvan como datos de entrada al modelo, pero la ventaja que tienen es que no son específicos, es decir, los mismos datos, pueden servir para solucionar distintos problemas.

En nuestro caso, al tener acceso a los datos en formato `pcap`, podemos decidir qué datos nos interesan, de entre una gran variedad de posibilidades.

Conversión de datos a csv Los algoritmos de aprendizaje automático como las redes neuronal usan una matriz como datos de entrada. Una de las formas más similares a las matrices, son las tablas `csv`, en la que la relación es evidente. Se van a tener que mutar los datos crudos a datos con formato `csv`, que luego leeremos con `python` para importar a nuestro modelo.

Sanado de dataset Es posible que el dataset generado tras convertir los datos de `pcap` a `csv` contenga errores. Un ejemplo de error es que cualquier campo no tenga valor. Si se detecta este error, debemos arreglarlo, ya que, aunque las redes neuronales solventan este error, es mejor que todos los campos tengan un

valor. En este caso, el valor nulo lo cambiamos a `UNKNOWN`. Todos estos errores deben ser contemplados y arreglados.

En este dataset en concreto, los pasos a realizar para sanar el dataset son:

- Eliminar las comas extra introducidas por la herramienta de extracción de "features"
- Eliminar los valores nulos en el dataset
- Eliminar las tuplas que contienen información corrupta, en este caso, ha habido un problema con la herramienta de extracción de features `tshark`, en la que ha introducido dos veces las columnas de `ip.src` e `ip.dst`.

Los pasos de sanado se encuentran en el script `sanitize.py` incluido en la entrega de la práctica.

Determinar datos categóricos y continuos Las variables categóricas son las que pueden obtener valores concretos, predefinidos dentro de una serie de posibilidades.

Las variables continuas, son las que pueden obtener valores infinitos, es decir, existe una infinidad de valores continuos. Estas variables suelen ser numéricas.

En nuestro caso, las variables categóricas son:

- `ip.src`
- `ip.dst`
- `_ws.col.Protocol`
- `_ws.col.Info`

La variable continua que tenemos es:

- `frame.len`

Codificar datos categóricos y continuos Tanto las variables categóricas como las continuas, se deben codificar de una forma especifica para que la red neuronal interprete los valores correctamente.

Las formas más populares de codificar las variables categóricas es con encodeado `one-hot`, mientras que las continuas, se pueden encodear normalizando sus valores.

Dividir el dataset en entreno y test Es buena práctica subdividir el dataset en dos.

Estas dos secciones serán la sección de entreno y la sección de prueba. El modelo se entrena con la sección de entreno, pero se reserva una sección, por lo general del 30% del tamaño del dataset para revisar el modelo al acabar la fase de entreno. Una de las razones por las que se hace esto, es para asegurarnos que nuestro modelo funciona correctamente. Esto es porque hemos usado unos datos para entrenar a nuestro modelo, pero estos datos, ya los ha visto, y el modelo se

ha configurado de acorde con estos. A nosotros nos interesa generar un modelo que sea capaz de decidir con datos que no se hayan usado nunca, ya que esta es su finalidad.

Resultados esperados del proceso

Cuando se acabe de entrenar el modelo, este nos proporciona un porcentaje de acierto, basado en los datos de test. Este sera siempre nuestro límite superior de probabilidad de acierto. Una vez estemos satisfechos con este valor, si satisface nuestros requisitos a nivel de red, podremos desplegar el modelo.

Una vez despleguemos el modelo en nuestro entorno de producción, debemos estar al corriente de los paquetes de red que marca como `Botnets`. Si implementamos un IDS, podemos programar un servicio que nos notifique mediante correo electrónico o incluso por un canal de Slack, una plataforma de gestión de trabajo.

Implementación

Programas utilizados

Para realizar este ejercicio, se ha contado con una plataforma creada por Google específicamente para realizar proyectos de estas características. Una vez realizado, se ha descargado y ejecutado en mi máquina local, para asegurar que todas las fases funcionan correctamente.

La limitación de la plataforma de Google es que, al usar la capa gratuita, te proporcionan una máquina con poca RAM y procesador. Por este motivo, se han tenido que hacer algunas modificaciones desde la descarga del archivo de la plataforma de Google a la plataforma local. Uno de los más evidentes, es el dataset. A la plataforma, estaba subiendo una muestra pequeña del original. Ahora, esto lo hago directamente desde el código.

En remoto Se usa un framework hecho por Google llamado colaboratory, que deja todo preparado para realizar análisis de datos para Aprendizaje Automático. Tiene la forma de los cuadernos Jupiter y permite tener código y texto en una misma vista.

En local, Programas/Ayudas utilizadas En esta práctica, se han usado varias tecnologías. Las más importantes son:

- Python
- Keras con TensorFlow
- Bash

En relación a Python, podíamos usar la versión 2 o 3 del mismo lenguaje de programación. Se ha decidido usar la versión 3 ya que es lo recomendado por la comunidad.

En relación a Keras, este es el framework más utilizado para codificar modelos de redes neuronales. Se describe su elección en más detalle en la sección: 'Elección del framework'.

En relación a Bash, es un lenguaje de scripting muy potente. Funciona como una capa de unión entre los programas que el usuario tiene incluidos en su "Path". Una de las utilidades más importantes del mismo es la posibilidad de vincular la salida de datos de un programa directamente con la entrada de datos del siguiente mediante el uso de "Pipes". Esto permite al usuario crear cadenas de flujo de datos de una forma muy sencilla y eficiente.

Tratado de datos *Desde el dataset a la ingesta en Python*

Desde el dataset plano, se realizan una serie de mutaciones sobre los datos para que sea más sencillo operar sobre ellos directamente en el script Python `modeloSecuencial.py`.

Estas alteraciones son:

- Conversión
- Saneado del dataset
- Preparado para modelo

Conversión Este proceso lo realiza el archivo bash llamado `prepareData.sh`. Este script convierte todos los datos de `pcap` a `csv`. Además, concatena todos los datasets con paquetes maliciosos en un dataset que contiene únicamente paquetes maliciosos y todos los datasets con tráfico legitimo en un solo dataset que contiene únicamente paquetes legítimos.

Saneado del dataset El proceso de saneado lo asume el script `sanitize.py` pasándole por parámetros los archivos que se quieran sanear.

Las acciones que realiza son:

- Eliminación de columnas sobrantes introducidas por el comando `tshark`
- Sustitución de valores nulos por predefinidos (`UNKNOWN`)
- Eliminar tuplas invalidas

En el caso de nuestro dataset, el programa `tshark` genera muchas entradas corruptas relacionadas con la dirección `IP 192.168.50.17`. El error se puede detectar analizando el campo `frame.len` en busca de una dirección `IP`. Algunas formas en las que se hubiese podido tratar las entradas corruptas son:

- Eliminado
- Acondicionado/Arreglado

En nuestro caso, analizando las entradas corruptas, se puede observar que cada una contiene los campos `ip.src` e `ip.dst` duplicados. Eliminado esta duplicidad, se hubiese podido contar con estas tuplas.

Se ha decidido eliminar las ocurrencias corruptas, debido a que tenemos mucha variedad y cantidad de datos.

Preparado para modelo El script `identifyAndsort.sh` se encarga de añadir una columna extra al dataset, para indicar al modelo que paquetes de red son los maliciosos y cuales los legítimos. Ahora que ya tenemos identificada cada tupla con su clase (Si es Botnet o no) podemos unir ambos archivos para posteriormente ordenarlos por el campo `frame.time_epoch`.

Realmente no es necesario ordenar el archivo, pero se ha decidido hacerlo de esta forma por tener un archivo con los datos ordenados. En el script `modeloSecuencial.py` ya vamos a obtener un extracto del archivo de forma aleatoria, asegurándonos una buena muestra.

Todos los archivos anteriores se encuentran agrupados en un script llamado `prepareData.sh` para agilizar el proceso.

Elección del framework

A la hora de generar el modelo, nos encontramos con varias opciones para llegar a la misma finalidad. Entre las opciones, tenemos:

- SciKit, un framework de aprendizaje automático
- PyTorch, un framework de aprendizaje profundo
- Keras, un framework de aprendizaje profundo

Viendo las opciones anteriores, decidimos usar Keras, debido a su versatilidad y abstracción de los algoritmos de aprendizaje profundo. Este framework nos puede proporcionar la potencia de varios frameworks de redes neuronales como TensorFlow, Theano o CNTK. Nosotros vamos a usar Keras en combinación con TensorFlow para generar el modelo.

Más concretamente, vamos a usar el modelo `sequential_model` para generar nuestra red neuronal.

Importación del dataset a Python

En este dataset conviven tanto datos categóricos como continuos.

Se ha tomado la decisión de no incluir los datos del `epoch_date` ya que no tienen mucha relevancia a la hora de entrenar la red neuronal.

Los datos categóricos debemos cambiarlos a datos `dummy` antes de que los use el algoritmo de generación del modelo. Para realizar este paso, usamos la codificación `one-hot`.

Los datos continuos, debemos normalizarlos, para eso, tenemos que asignar 0 al valor más bajo y 1 al valor más alto. Una vez tengamos el dataset tratado para ser ingerido por el modelo de la red neuronal, podemos empezar a entrenar, con un 70% del dataset, para posteriormente revisar con un 30% del dataset.

A excepción de la fase de tratado de datos y preparado del dataset, todas la demás fases, están en el archivo llamado: `modeloSecuencial.py`.

`modeloSecuencial.py` - una breve descripción Este script se encarga de:

- Leer una fracción del dataset de forma optimizada
- Preparar los datos de entrada
- Separar el nuevo dataset en datos de entrada y salida
- Subdividir el dataset en datos de entrada de prueba y entrenamiento y datos de salida de prueba y entrenamiento
- Definir el modelo
- Definir las capas de la red neuronal
- Compilar el modelo
- Entrenar el modelo
- Evaluar el rendimiento del modelo
- Guardar el modelo generado a disco

Al acabar la ejecución del script, este ha generado un modelo de red neuronal que es capaz de distinguir entre paquetes de red `Botnet` y paquetes de red corrientes.

Leer el `.csv` En este paso, se usa el método `read_csv` de `pandas`, pasándole los tipos de datos que se va a encontrar en cada columna, para optimizar más la carga del dataset. Además, se le dice al método el numero de linea en el que se encuentra el nombre de cada columna.

Preparar los datos de entrada para el modelo En esta sección, se codifican los datos categóricos y continuos en una matriz que el modelo es capaz de entender y utilizar.

Los métodos que se han usado para cifrar los datos son:

- *OneHotEncoder* de sklearn.preprocessing
- *Normalize* de sklearn.preprocessing

Durante la codificación del script, uno de los errores que había cometido era dividir el dataset en datos de entrenamiento y testeo antes de preparar el dataset. El error con el que me estaba encontrando era que el tamaño del dataset de entrada para revisar el modelo no era el mismo que el tamaño de el dataset de entrenamiento del modelo, por tanto, no se podía revisar el modelo con los datos de prueba.

Al darme cuenta de este fallo, se convierten previamente todos los datos de entrada al modelo antes de subdividir el dataset en datos de entrenamiento y datos de testeo.

separar el dataset en datos de entrada y salida Para poder entrenar nuestro modelo de redes neuronales (Supervisado), necesitamos poder decirle

al algoritmo cuando ha acertado en la predicción y cuando esta predicción es incorrecta.

La forma más común de almacenar esta información es incluir la clase `Botnet` de cada tupla en la ultima columna de la misma. De esta forma, tenemos un dataset independiente, que no necesita nada más para poder ser útil para entrenar.

Para entregar al modelo los datos de entrada (`X`), tenemos que dividir el dataset generado, para que no incluya la clase `Botnet`, ya que esta es la salida que tiene que generar el mismo y usar este como entrada.

Generamos el dataset de salida (`y`) seleccionando únicamente la clase Botnet.

Subdividir el dataset en datos de entreno y testeo Subdividimos el dataset en datos de entrenamiento y testeo para cada tipo.

Se usa el método `train_test_split` de `sklearn` para subdividir el dataset en datos de entrenamiento y datos de prueba. Se hace esto porque solo de esta forma podemos averiguar si el modelo generado no ha sobreaprendido. Sobre aprender, en redes neuronales significa que el modelo ha memorizado cada tupla, de forma que se ha vuelto demasiado específico para nuestro cometido.

Si un modelo ha sobreaprendido, no sabe distinguir tráfico con el que no haya entrenado. Para nuestra finalidad, esto es poco beneficioso, ya que el modelo se debe encargar de filtrar el tráfico de Botnet, lo haya visto anteriormente o no.

Definición del modelo De los posibles dos modelos proporcionados por Keras, se ha optado por implementar el modelo secuencial.

Capas del modelo Hay muchos tipos de capas disponibles en Keras. De entre las capas Core:

- Dense
- Activation
- Embedding
- Masking
- Lambda

Se ha optado por usar las capas `Dense`. Esta capa es la más básica de todas, pero bastará para clasificación que queremos hacer.

A la primera capa se le pasa la dimensión de los datos de entrada, que se obtiene dinámicamente según el dataset de entrada `X_train`.

Compilado del modelo El modelo se compila con el comando:

```
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

Entrenamiento del modelo Cuando modelo esta definido y compilado, se tiene que entrenar, para generar las predicciones. El comando que se usa es:

```
model.fit(X_train, y_train, epochs=15, batch_size=X_train.shape[1],
verbose=2)
```

Los parámetros se explican en la fase de pruebas y resultados.

Evaluación de rendimiento del modelo El comando que se usa para evaluar el modelo seleccionado es:

```
_, accuracy = model.evaluate(X_test, y_test, verbose=0)
```

En esta parte es donde realmente se revisa que el modelo ha aprendido correctamente y no ha sobreaprendido. Al revisar el modelo con datos completamente nuevos para el, podemos asegurar su correcto entrenamiento.

Este es el paso que determina si el modelo esta listo para ser desplegado a producción. Una vez haya cumplido con los margenes de error estipulados con anterioridad, podemos desplegarlo con confianza.

Guardado del modelo a disco Para poder desplegar el modelo, tenemos que poder guardarlo en el disco. Este es un archivo que contiene la información que ha sido generada en la fase de entrenamiento (`model.fit`).

El comando que ejecuta esta acción es:

```
model.save('modeloSecuencial.h5')
```

Que, almacena el archivo llamado `modeloSecuencial.h5` a nuestro directorio base.

Pasos de ejecución

Vamos a establecer el directorio principal desde el cual trabajaremos durante todo el ejercicio. De ahora en adelante, esta sera la carpeta base de esta práctica. (`~/`) Este se llama:

```
TrabajoFinal
```

1. Configuración del espacio de trabajo Descargamos el dataset desde la siguiente URL: ISOT HTTP Botnet Database a nuestro directorio raíz.

Extraemos el dataset en este directorio, de forma que se crea un directorio llamado `isot_app_and_botnet_dataset`.

2. Tratado de datos Copiar los scripts:

- `extraction.sh`
- `sanitize.py`
- `identifyAndsort.sh`

- `prepareData.sh`

Al directorio llamado `isot_app_and_botnet_dataset`.

1. Una vez tenemos estos archivos en nuestro directorio, procedemos a **ejecutar el `script extraction.sh`.**
2. Seguimos **ejecutando el script `sanitize.py`** con el nombre de los archivos que queremos sanear. En nuestro caso, `merged_network_benign_traffic.csv` y `merged_network_malign_traffic.csv`.
3. Para finalizar, **ejecutamos el comando identifyAndsort.sh**

Como alternativa, se ha preparado un script que engloba los pasos anteriores en uno, para directamente poder importar con el script del modelo `modeloSecuencial.py`. Simplemente **Ejecutamos prepareData.sh**, se ejecutan todos los scripts anteriores.

Limitaciones/restricciones en la implementación

A día de hoy, no se pueden filtrar los paquetes que pasan por la red con este modelo, debido a que falta una capa más de implementación.

Una de las limitaciones a la hora de realizar esta implementación ha sido la ingente cantidad de datos que contiene el dataset. El archivo de texto plano que contiene todos los paquetes del dataset tiene en torno a diez millones de entrada, esto son en torno a 10 millones de datos sobre paquetes de red. Para analizar todos estos datos, al menos de la forma en la que se ha procedido en esta ocasión, se hubiese necesitado un ordenador muy capaz, con más de siete TiB de memoria.

No haber visto el modelo de forma gráfica, para poder entender mejor los datos siendo tratados. Al poder ver el modelo de forma gráfica, es más sencillo decidir el numero de capas ocultas que necesita el modelo, pero sin esta información, la mayor parte de esa decisión ha sido prueba y error informados.

Pruebas y resultados

Proceso a seguir para obtener los datos

1. Descarga de los archivos desde mi repositorio de GitHub: sergiorosello o desde la entrega proporcionada a través de AlF.
2. Descarga del dataset desde ISOT HTTP Botnet Database
3. Copiar todos los archivos excepto `modeloSecuencial.py` a la carpeta extraída.
4. Ejecutar el script proporcionado en la entrega llamado `prepareData.sh`.

Al llegar a este punto, se habrá generado un archivo llamado `sorted_merged_traffic.csv`. El programa de Python en el que se encuentra el modelo se encarga de extraer una muestra de los datos dentro del dataset, ya saneado.

Aplicación de los programas/scripts

El script en el que se define el modelo y se entrena se llama `modeloSecuencial.py`. Para **ejecutar este archivo, se puede usar el comando `python modeloSecuencial.py`**

Una vez haya terminado, este script habrá generado un modelo capaz de discriminar entre paquetes de red `Botnet` y benignos.

Casos comprobados y valores de datos iniciales y de parámetros

Modelo Se ha usado el modelo Secuencial, ya que proporciona la flexibilidad que se necesita y la facilidad de uso. En esta primera práctica, me ha parecido interesante empezar desde la base.

En futuras implementaciones, sin duda empezare a usar el API Funcional.

Función de activación Se han usado dos funciones de activación desde el principio de la práctica. Las funciones de activación determinan si una neurona cumple con los requisitos para activarse. Si sobrepasa la función de activación, esta neurona se activa y el flujo continua a través de esa neurona por la red neuronal.

Los dos tipos de funciones de activación que se han usado son:

- `ReLu`
  - Es una mejora a la función de activación binaria.
  - Es sencilla de aplicar, por tanto es rápida, buena para una primera capa
  - Como desventaja, puede generar neuronas "muertas"
- `sigmoid`
  - Devuelve siempre un valor entre 0 y 1
  - Valores gradualmente más altos, devuelven valores más próximos a 1 y viceversa

Optimizadores Existen siete optimizadores distintos que el analista puede usar a la hora de compilar el modelo de red neuronal. Entre estos, el optimizador que se ha decidido usar es `adam`, debido a que según kingma et al., 2014 "Es computacionalmente eficiente, no necesita mucha memoria, invariante al reescalado diagonal de gradientes y es adecuado para problemas que tienen muchos datos/parámetros."

Métricas Cuando se ha compilado el modelo, se ha usado la métrica "accuracy" para saber el numero de veces que el modelo predice el "label". Esto es: Saber la frecuencia con la que el modelo acierta determinando el tipo de tráfico que se ha comprobado.

Resultados obtenidos e interpretación de los datos

En la primera prueba que se han realizado:

- **Modelo:** Secuencial
- **Capas:**
  - Dense, 32 neuronas de salida, activación ReLu, kernel_initializer he_normal
  - Dense, 1 neurona, activación sigmoide
- **Compilacion:**
  - Loss binary_crossentropy, optimizador adam, metrics accuracy
- **Entreno:**
  - Epochs 10
  - batch_size 16

Se ha llegado a una precisión del 96% en el ultimo 'epoch' del entrenamiento, pero se ha obtenido un 94% de acierto usando el modelo con los datos de test. Estos datos son bastante buenos.

A partir de estos datos se ha procedido a cambiar los parámetros de la red neuronal para ajustarla mejor. (Desde función de activación, Optimizadores, nuevas capas, de distintos tipos)

Se han realizado pruebas para determinar el numero de capas ocultas necesarias. Para conseguir esto, se ha añadido una nueva capa oculta, con 64 salidas, pero los resultados no han variado mucho.

Entiendo por este experimento que no es necesario añadir capas ocultas, porque el dataset con el que estamos operando se puede dividir de forma lineal.

El máximo porcentaje de acierto que se ha conseguido ha sido 96% en fase de prueba. Este porcentaje se ha conseguido con los datos presentados a continuación.

Este ejercicio se ha llevado tan lejos como dejar solamente una neurona de salida en la capa de entrada y una salida en la capa de salida. Aun con estas características, la red neuronal ha obtenido una media de 95% acierto.

- **Modelo:** Secuencial
- **Capas:**
  - Dense, 1 neuronas de salida, activación ReLu, kernel_initializer he_normal
  - Dense, 1 neurona, activación sigmoide
- **Compilacion:**
  - Loss binary_crossentropy, optimizador adam, metrics accuracy
- **Entreno:**
  - Epochs 10
  - batch_size 160

Una buena forma de probar nuevas estructuras de capas y neuronas es siguiendo

la siguiente regla:

Si se añaden más capas ocultas, proporcionas a la red neuronal la habilidad de discriminar figuras más complejas (Construyendo sobre las segmentaciones lineales básicas) Si se añaden más neuronas dentro de una misma capa oculta, se añaden más características, por tanto, más formas básicas de separar la información.

Teniendo en cuenta que nuestra red neuronal es sencilla, porque cuenta con una salida: Si es `Botnet` o no, mi teoría es que la forma de subdividir la red neuronal para separar estas características es sencilla.

Revisando ahora los hiperparametros:

Teniendo en cuenta que 'sample' es cada array, en este caso unidimensional de datos de entrada al modelo.

- epochs (Numero de veces que el algoritmo se ejecuta)
- Batch_size (Numero de 'samples' que tiene que procesar la red hasta que se computen los cambios a realizar)

Se ha probado a reproducir el modelo anterior con 10 'epochs' y `X_train.shape[1]` `batch_size` pero con el mismo numero de 'epochs', no ha llegado a un porcentaje de precisión tan alto. De hecho, parece que es el punto justo en el que se empieza a asentar. Menos `epochs` harían que el modelo no fuese lo preciso como podría ser, unas cuantas mas, nos ayudarían a saber que tenemos el modelo más preciso dados los datos de entrada que podemos generar.

Por este motivo, se ha decidido dejar los hiperparametros `epoch` a 15 y `batch_size` a `X_train.shape[1]` para generar un modelo de aprendizaje llamado 'batch gradient descent'.

La red neuronal resultante, dadas las pruebas realizadas queda de la siguiente forma:

- **Modelo:** Secuencial
- **Capas:**
  - Dense, 1 neurona de salida, activación ReLu, kernel_initializer he_normal
  - Dense, 1 neurona, activación sigmoide
- **Compilacion:**
  - Loss binary_crossentropy, optimizador adam, metrics accuracy
- **Entreno:**
  - Epochs 15
  - batch_size `X_train.shape[1]`

Este modelo ha generado un porcentaje de acierto del 94.32% en la fase de prueba.

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1)                 678
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 2
=================================================================
Total params: 680
Trainable params: 680
Non-trainable params: 0
_________________________________________________________________
Compiling the Keras model
Fitting the Keras model
Epoch 1/15
 - 1s - loss: 0.6967 - accuracy: 0.0812
Epoch 2/15
 - 1s - loss: 0.6957 - accuracy: 0.4601
Epoch 3/15
 - 1s - loss: 0.6946 - accuracy: 0.4885
Epoch 4/15
 - 1s - loss: 0.6934 - accuracy: 0.5074
Epoch 5/15
 - 1s - loss: 0.6923 - accuracy: 0.5819
Epoch 6/15
 - es - loss: 0.6911 - accuracy: 0.6698
Epoch 7/15
 - 1s - loss: 0.6900 - accuracy: 0.8539
Epoch 8/15
 - 1s - loss: 0.6889 - accuracy: 0.9269
Epoch 9/15
 - 1s - loss: 0.6878 - accuracy: 0.9499
Epoch 10/15
 - 1s - loss: 0.6867 - accuracy: 0.9540
.
. (Comentado por brevedad)
.
Epoch 15/15
 - 1s - loss: 0.6815 - accuracy: 0.9540
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1)                 678
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 2
=================================================================
Total params: 680
```

```
Trainable params: 680
Non-trainable params: 0

_________________________________________________________________
evaluating the Keras model
Accuracy: 94.32
Model saved to disk
```

Otros valores a revisar

En esta práctica se ha usado el modulo `sequential_model`, debido a que es una red neuronal sencilla, pero seria muy interesante entrenar el modelo con modelos más avanzados, que contengan neuronas compartidas entre capas, distintas, entradas y salidas por capa o grafos de capas.

La fase de elección y tratado de datos de entrada para el modelo es una de las más importantes a la hora de realizar un buen modelo. Por este motivo, me hubiese gustado poder variar los datos de entrada. Como ejemplo, en vez de usar un extracto proporcional de datos de entrada (Teniendo en cuenta que hay mucho más tráfico de botnets que convencional) usar datos de entrada que contengan el mismo numero de paquetes de Botnet como de paquetes de red convencionales.

Creo que de esta forma, el modelo hubiese tenido más variedad de información. Quizá, una posible consecuencia es esto es que hubiese salido más general. Entiendo que no, porque los datos que se han seleccionado del dataset, no han sido de un mismo Botnet, sino un extracto de todos los datos. De todas formas, me quedo con la duda.

Durante la realización de esta práctica, ha sido muy complicado justificar las características de las capas internas de la red neuronal. Seguramente debido a la necesidad de profundizar más mi conocimiento en la materia y a la falta de documentación de la misma. El problema con el que me he encontrado es que, entendiendo todos los factores que entran en juego con la síntesis de una red neuronal, ha sido muy complicado tomar decisiones informadas sobre la estructura de la misma.

Una solución a este problema es dejar que la red neuronal se entrene a si misma. Esta idea tiene nombre de 'Neural Architecture Search' y existe un framework que se encarga de generar una red neuronal optima dado los datasets de entrada y salida. Este framework se llama autokeras

Posibles mejoras

El dataset ha sido generado en una misma red. Cada máquina ha estado enviando tráfico específico de un malware concreto. Esto es beneficioso, porque podemos identificar a simple vista (Según la IP) si ese tráfico es malicioso o no. El inconveniente que introduce este método es que no es tráfico de red verdadero,

ya que en la vida real, una sola máquina genera tráfico tanto normal como malicioso.

Otro inconveniente de la forma en la que se ha capturado el dataset es que los datos malignos se han capturado antes que los normales, haciendo que, si se ordena todo el dataset, todo el tráfico de red esté segmentado por naturaleza.

Un problema persistente a lo largo de toda la práctica ha sido la enorme cantidad de datos con los que se trabajaba. Desde la adquisición de los mismos, pasando por su gestión, hasta su utilización con el modelo de red neuronal.

Sería muy interesante utilizar el modelo generado para que filtre el tráfico en tiempo real. Se ha realizado una breve investigación y se podrían concatenar herramientas como mitmproxy, haciendo uso de su API para obtener los datos necesarios para convertir los paquetes en datos que necesita nuestro modelo. A continuación, podemos pasar la tupla por el modelo generado para realizar nuestra predicción, si es Botnet o no. Según el resultado del modelo, dejamos que pase el paquete por nuestro proxy.

Una de los inconvenientes que le encuentro al planteamiento anterior es que no podríamos codificar los datos categóricos ni contínuos como lo hemos estado haciendo hasta ahora, porque no tenemos todos las posibles valores que puede cobrar una propiedad.

Comentarios sobre la realización de la actividad

La actividad ha sido un ejercicio completo de tratado de datos para un propósito concreto. Ha sido muy interesante pasar por todas las fases a las que se enfrenta un analista de datos. Sobre todo, darme cuenta que el manejo de datos y su correcta codificación consumen más tiempo incluso que implementar el modelo. Más aún si el analista tiene claro qué modelo debe implementar para solucionar el problema. En mi experiencia, esta parte ha sido la más incierta. Sabia los pasos necesarios para adaptar los datos al modelo, pero en el caso de la red neuronal, estos no han sido tan evidentes.

Bibliografía

- *Neural networks*:
  - `https://scikit-learn.org/stable/modules/neural_networks_supervised.html`
- *Dataset*:
  - Alenazi A., Traore I., Ganame K., Woungang I. (2017) Holistic Model for HTTP Botnet Detection Based on DNS Traffic Analysis. In: Traore I., Woungang I., Awad A. (eds) Intelligent, Secure, and Dependable Systems in Distributed and Cloud Environments. ISDDC 2017. Lecture Notes in Computer Science, vol 10618. Springer, Cham
- *one_not encoding*

  - https://scikit-learn.org/stable/modules/ generated/sklearn.preprocessing.OneHotEn
- *Categorical_functions*
  - https://keras.io/api/utils/python_utils/#to_categorical-function
- *read_csv*
  - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
- *Keras sequential model*:
  - https://keras.io/guides/sequential_model/
- *Keras training and evaluation*:
  - https://keras.io/guides/training_with_built_in_methods/
- *Keras functional model*:
  - https://keras.io/guides/functional_api/
- *Keras layers*:
  - https://www.tutorialspoint.com/keras/keras_layers.htm
- *Teoria de capas ocultas*
  - https://datascience.stackexchange.com/questions/26597/how-to-set-the-number-of-neurons-and-layers-in-neural-networks