Anexo 3.2.1 Manual SKYNNET

CONSORCIO:

NOKIA

ORGANISMOS DE INVESTIGACIÓN Y COLABORADORES:

Ciemat
Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas

SONRYE fotografía

Proyecto cofinanciado por Red.es, nº de expediente IDI-2021/C005/00144167

1. Introducción

Este documento es el manual de instrucciones de SkyNNet que comprende la deconstrucción de una red neuronal y su posterior ejecución de manera distribuida y paralela.

Este proceso se compone de dos pasos

1. Deconstrucción de redes neuronales con la herramienta *sk_tool* que produce el código deconstruido para su ejecución secuencial
2. Ejecución distribuida y paralela de la red deconstruida con *SkyNNet*, que distribuye el código deconstruido en varios agentes que pueden estar en distintos computadores.

El primer paso se puede ejecutar directamente en la herramienta *SkyNNet* pero conceptualmente son dos pasos (deconstrucción y ejecución paralela y distribuida)

1.1. Instalación

Descargar el repositorio de SkyNNet de github: https://github.com/SKYNNET-ORG/SKYNNET

Este repositorio contiene los siguientes elementos:

- Maker: Este componente se encarga de preparar el código de entrada para convertirlo en código distribuible y paralelizable, además incluye la opción de ejecutar el mismo la deconstrucción de redes neuronales.
- Agente: El agente que ejecuta el código de cada proyecto, también incluye el dashboard para controlar la ejecución o los elementos para que desde el dashboard sean controlados remotamente.
- Deployer: Este componente actúa de intermediario entre el agente y el maker, se encarga de asignar el código producido por el maker a los distintos agentes, además tiene opciones de vigilancia del proyecto para mejorar el desempeño.

Estos componentes se pueden descargar y ejecutar en cualquier ubicación pero para ejecutar los proyectos hay que realizar el siguiente paso:

- Crear carpeta llamada cloudbook en el siguiente directorio C:\Users\(usuario).Quedando la siguiente ruta C:\Users\(usuario)\cloudbook, en esta ruta deberá ir cada carpeta de cada proyecto, que es la que contendrá el código que se deconstruye y ejecutará de manera paralela y distribuida.
- Dentro de cada carpeta de proyecto se crearán las siguientes subcarpetas:
    - agent: Con información local sobre el agente, como potencia cedida de manera cualitativa o dirección ip del mismo.
    - distributed: Esta carpeta es la que se usará en un sistema de ficheros distribuido, y todos los agentes compartirán, contiene el código asignado (salida del maker), directorio de trabajo, y información sobre el resto de agentes y ejecución.
    - original: Con el código fuente original, la entrada del maker
    - **A la carpeta del proyecto nos referiremos en este documento como el working dir**

1.1.1. Entorno Anaconda

Falta nombre del fichero correcto, en SKYNNET hay 2

La configuración de entorno de anaconda se encuentra en GIT. Fichero skynnet.yaml del repositorio. Para instalar el entorno hay dos maneras:

1. Por interfaz → dando al botón de import/importar, seleccionando después la ruta. La siguiente foto muestra el primer botón, luego siga las instrucciones de la interfaz.

2. Por comandos → ejecutar el siguiente comando: **conda env create -f archivo.yaml**, el archivo .yaml debe estar en la ruta donde ejecutas el comando, o puedes escribir la ruta completa hacia el mismo.

1.1.2. Dependencias del proyecto

Cuando estén las dependencias de skynnet en el yaml, se copian y pegan aqui

Si no se usa anaconda, a continuación se encuentran las dependencias del proyecto:

- versión de python
- dependencias de pip

2. Preparación SKYNNET

Para ejecutar un programa en skynnet se debe integrar en dos fases:

- Adaptación a SKTOOL
- Adaptación a SKYNNET

A continuación se describen ambas fases:

2.1. Adaptación a SKTOOL

Tal y como se describe en el manual **A.3.1 Manual SK_TOOL** , los pasos más importantes a tener en cuenta para adaptar el código fuente original son:

- imports y carga de datos: Los imports tienen que ser de un módulo por línea, los datos no se pueden cargar en forma de tupla,Antes de la carga de datos se debe incluir la etiqueta de Skynnet de ejecución distribuida #CLOUDBOOK:NONSHARED
- etiqueta skynnet: Tipo de problema que se quiere resolver: Regresión, clasificación multiclase, clasificación binaria, y medida de la red deconstruida que se quiere obtener
- variables: Las variables de datos, son las variables de entrenamiento, validación (si no usas un *validation_split* serían _DATA_VAL_X/Y) y test. El resto de variables son referidas a las capas de la red neuronal, por ejemplo las variables _NEURON_ son las capas ocultas de un MLP, el _NEURON_ con un número más alto.

```
sk_vars_list = ["_EMBEDDING_","_NEURON_","_CELL_","_EPOCHS","_FILTERS_"]
```

- modelo a deconstruir: Al definir el modelo se recomienda usar el API funcional de Keras, y al definir las capas del modelo, usar las variables específicas para deconstruir
- predict original : no debe ser elemento por elemento si no por lote completo, ya que la transformación que hace sktool de dicho predict a la predicción compuesta así lo requiere .
- Tiene que haber una función llamada main en el código, que contenga el punto de entrada al programa, en caso de no haberla porque no se necesite, se tiene que escribir aunque sea vacía

2.1.1. Recopilación requisitos y consejos de programación

- Antes de deconstruir comprueba en la tabla de deconstrucciones para tu tipo de problema
- La red que quiera deconstruir se tiene que meter entre las etiquetas de skynnet (tanto la creación como el entrenamiento y el uso/predicción)
- La etiqueta Skynnet del código es como sigue:
  SKYNNET:BEGIN_[REGRESSION|MULTICLASS|BINARYCLASS]_[ACC][LOSS]
  #Resto del código
  SKYNNET:END
- Solo se deconstruyen las variables definidas para sk_tool (_NEURON_,_EPOCH_,_EMBEDING_, etc...). Estas variables son las que se usan para construir la red, en una capa Dense de 400 neuronas, se hará _NEURON_1 = 400, y al crear la capa dense en lugar de un 400, se pondrá _NEURON_1.

  El resto de variables son para cada tipo de capa, pero su uso es el mismo, en lugar de poner un numero, ese numero se asigna a la variable correspondiente y en la capa se escribe la variable:

  - _EMBEDDING_ Para redes como transformer que usan embeddings al construir la red
  - _FILTERS_: Para redes convolucionales, que usan capas con filter, en lugar de poner un número
  - _CELL_: Variable para redes recurrentes, que usan la variable cell en algunos tipos de celda
  - _EPOCHS_: Variable que indica las épocas, no va en ninguna capa, es de la función fit

- Para los datos se usan las siguientes variables: Por uniformidad y para que la herramienta siempre conozca los datos con los que tiene que tratar, se indican los datos que se usen de la siguiente manera:

  _DATA_TRAIN_X: Datos de entrenamiento
  _DATA_TRAIN_Y: Etiqueta de los datos de entrenamiento
  _DATA_TEST_X: Datos de test
  _DATA_TEST_Y: Etiqueta de los datos de test
  _DATA_VAL_X: Datos de validación
  _DATA_VAL_Y: Etiquetas de los datos de validación
  Estas variables son las que se tienen que usar en las funciones *fit* y *predict*

- Si en la tabla de deconstrucción se indica que para tu problema no se reducen las épocas, no uses la variable _EPOCHS ya que las variables de sk_tool son solo para deconstruir, y en este caso no se quiere deconstruir.
- **Requisito obligatorio:** Tiene que haber una función llamada main en el código, que contenga el punto de entrada al programa, en caso de no necesitarla, se tiene que escribir aunque esté vacía.
- **Restricción importante: El nombre de la red que creas, ha de ser el mismo que el nombre de la red que haces predict**, aunque por el camino hayas invocado varias funciones y usado distintos alias: Esto es porque la herramienta sk_tool al analizar la red asigna el nombre con el que se crea, y al analizar el código de manera estática no puede conocer los alias de la red sobre la que haces predict.
- **Es muy importante que tengas en cuenta los nombres de variables de skynnet** (_DATA_TRAIN, _NEURON, _EMBEDDING_, etc...) para invocar a tus funciones y a las funciones de la red, porque son estos datos los que indican la deconstrucción y división de datos, si has usado otros nombres, estos no se van a tener en cuenta en la deconstrucción.
- **Atención**: La neuron con el número más alto es decir _NEURON_X (siendo X el número más alto de todas las variables _NEURON_), se toma como el número de categorías totales en los problemas de clasificación, es la que se debe poner en la última capa de un clasificador.
- Si pretendes usar el código resultante con skynnet, no hace falta que incluyas ninguna etiqueta de cloudbook, ya se encarga la herramienta de incluirla
- Como la herramienta genera automáticamente un punto de entrada para invocar las n subredes, los conjuntos de datos de entrenamiento y test, se tienen que proporcionar enteros. Si tu programa no puede permitir esto, se pueden proporcionar como funciones o como generadores, ver explicación en el capítulo de código de entrada avanzado (apartado 5 de este manual)
- **Requisito**: Mantén la indentación de tu código de manera coherente, es decir si en un bloque estas tabulando con 4 espacios, no hagas un bucle for tabulado con 2, cada bloque de código ha de mantener la misma tabulación ya que el código de salida va a producir una tabulación de 4 espacios uniforme para todo el código, y si el código de entrada no es uniforme, va a producir errores en la salida.

2.1.2. Tabla resumen de estrategias de deconstrucción

La siguiente tabla pertenece al documento E44 y se ha obtenido tras efectuar un conjunto de pruebas masivas de deconstrucción de diferentes modelos neuronales. Esta tabla corrobora y mejora las estrategias de deconstrucción alcanzadas en la primera fase del proyecto y han sido implementadas en la herramienta SKTOOL que es parte del sistema skynnet

La siguiente tabla resume todas las conclusiones de clasificacion y regresion

	clasificación binaria	**clasificación multiclase**	**regresión local**	**regresión no local**
F	NA	decimal	entero	entero
subredes	2	$= \frac{\left(\frac{N}{(n/2)}\right)!}{2\left(\frac{N}{(n/2)}-2\right)!}$	F	F
reducción dim input	NO	NO	1/F	NO

reducción dim output	NO	1/F categorías	1/F	1/F
salida compuesta	se escoge la de mayor “certeza” para cada input	se obtiene multiplicando las categorías de las subredes	se obtiene concatenando las salidad de las subredes	se obtiene concatenando las salidad de las subredes
reduccion datos	1/2	1/F categorías => datos/F	NO	NO
reducción épocas	NO	con conv: NO solo densas: SI	NO	NO
reducción capas densas	neurons/F	neurons/F	neurons/F	neurons/F
reducción capas conv.	filtros/F	filtros/F	filtros/F	filtros/F
reducción dim. embedding	dim/F	dim/F	dim/F	dim/F
training time	≈ (tiempo original) / Num Subredes	≈ (tiempo original) / Num Subredes	≈ (tiempo original) / Num Subredes	< (tiempo original) / Num Subredes
reduccion modelo	[1/F, F]	[1/F, F]	NO	≈ 1/F
reducción ram operativa	[1/F, F]	[1/F, F]	IF F<8 , ≈ 1/F else [1/F, F]	IF F<8 , ≈ 1/F else [1/F, F]

2.2. Adaptación a SKYNNET

Un programa adaptado a sk_tool se puede usar directamente en SkyNNet como se explica en el apartado 2.2.2 de este documento, pero se va a explicar en este capítulo la estrategia de carga de datos, para que los datos que se carguen, sean compatibles con una ejecución distribuida en varios computadores, sin que el programador tenga que cambiar nada ni pensar en el entorno distribuido.

2.2.1. Carga de datasets en entornos distribuidos

La carga de datos en los programas que usen skynnet tiene que ser compatible tanto con el uso normal del programa como con el uso deconstruido y dividiendo los datos en distintas subredes, y con el uso distribuido teniendo los datos en distintas máquinas, a continuación se muestran estrategias de carga de datos que permiten la compatibilidad con estos tres usos, permitiendo al

usuario que su programa original se pueda deconstruir y ejecutar de manera paralela y distribuida sin tener que hacer cambios al código original o tener en cuenta el entorno paralelo y distribuido.

Como se cuenta en el documento A.3.1, los datos que se van a usar se tienen que declarar como variables nonshared y dentro de la etiqueta skynnet se asignan a las variables correspondientes

```
#__CLOUDBOOK:NONSHARED__
mnist = tf.keras.datasets.mnist
x_train = mnist.load_data()[0][0]
y_train = mnist.load_data()[0][1]
x_test = mnist.load_data()[1][0]
y_test = mnist.load_data()[1][1]

#SKYNNET:BEGIN_MULTICLASS_ACC_LOSS
_DATA_TRAIN_X = x_train
_DATA_TRAIN_Y = y_train
_DATA_TEST_X = x_test
_DATA_TEST_Y = y_test
```

Esta es la forma más sencilla de cargar los datos, las estrategias de carga de datos dependen de dos factores, la ubicación de los datos (local en el computador o externa en la nube) y la función de carga de datos (local en el código fuente, o externa en otro script al que se hace import).

- Según la ubicación de los datos:
  - Externa: No hay que hacer nada, solo los imports correctos
  - Local: Al usar Skynnet, los datos han de estar en el directorio de trabajo del working dir, es decir, en la ruta *.../cloudbook/proyecto/distributed/working_dir/datos*
- Según la función de carga de datos:
  - Externa: No hay que hacer nada, solo los imports correctos
  - Local: La función de carga de datos se tiene que etiquetar de la siguiente manera: #__CLOUDBOOK:LOCAL__, para que esté en todos los computadores distribuidos. Hay que invocar a la función de carga de datos de una manera específica.
    - En el main del programa, si no se tiene, hay que crearlo solo con la carga de datos e invocarlo antes del código de la red neuronal, es decir antes de la etiqueta Skynnet
    - Al principio de la etiqueta Skynnet, antes de indicar las variables con los datos
    - Las variables de datos declaradas como nonshared se tienen que rellenar en la función de carga de datos

A continuación se muestran ejemplos de la carga de datos (las rutas mostradas son relativas al working dir, es decir *C:users/user/cloudbook/proyecto/...*):

Datos externos y función de carga externa:

```
import tensorflow as tf, numpy as np
```

```
#__CLOUDBOOK:NONSHARED__
mnist = tf.keras.datasets.mnist
x_train = mnist.load_data()[0][0]
y_train = mnist.load_data()[0][1]
x_test = mnist.load_data()[1][0]
y_test = mnist.load_data()[1][1]

#SKYNNET:BEGIN_MULTICLASS_ACC_LOSS
_DATA_TRAIN_X = x_train
_DATA_TRAIN_Y = y_train
_DATA_TEST_X = x_test
_DATA_TEST_Y = y_test
```

Simplemente se hace el import de tensorflow y se accede a las funciones necesarias.

Datos externos y función de carga local:

```
import tensorflow as tf, numpy as np

#__CLOUDBOOK:NONSHARED__
x_train = None
y_train = None
x_test = None
y_test = None

#__CLOUDBOOK:LOCAL__
def load_data():
    global x_train
    global y_train
    global x_test
    global y_test
    x_train,y_train,x_test,y_test = tf.keras.datasets.mnist.load_data()

#__CLOUDBOOK:DU0__
def main():
    load_data()

#SKYNNET:BEGIN_MULTICLASS_ACC_LOSS
load_data()
_DATA_TRAIN_X = x_train
_DATA_TRAIN_Y = y_train
_DATA_TEST_X = x_test
_DATA_TEST_Y = y_test
```

Como los datos son externos, solo hace falta el import correcto y llamar a la función correspondiente.

Como la función de carga es local, se tienen que declarar las variables de datos como None de manera global, se asignan correctamente en la función load_data() ya que tienen que estar en una función para ejecutar en SkyNNet (si están en el código a nivel de módulo, el maker ignora esas invocaciones, porque solo traduce funciones e invocaciones a las mismas)

A esta función **hay que invocarla obligatoriamente en el main, y en la etiqueta skynnet** (esto es para que en el entorno distribuido, el agente 0, que puede no ejecutar funciones paralelas pueda tener los datos gracias a la invocación del main, y el resto de agentes que ejecutan funciones paralelas tengan todos los datos gracias a la invocación dentro de la etiqueta skynnet)

Datos locales y función de carga externa:

```
import load_datos

#__CLOUDBOOK:NONSHARED__

x_train = load_datos("./datos")[0][0]
y_train = load_datos("./datos")[0][1]
x_test = load_datos("./datos")[1][0]
y_test = load_datos("./datos")[1][1]

#SKYNNET:BEGIN_MULTICLASS_ACC_LOSS
_DATA_TRAIN_X = x_train
_DATA_TRAIN_Y = y_train
_DATA_TEST_X = x_test
_DATA_TEST_Y = y_test
```

Los datos están en la ruta *./datos* que es local al invocar el script principal, y al ejecutarse en skynnet tendrá que estar en la ruta: */distributed/working_dir*

Como la función de carga de datos es externa, solamente hay que hacer el import correcto.

Datos locales y función de carga local:

```
#__CLOUDBOOK:NONSHARED__
x_train = None
y_train = None
x_test = None
y_test = None

#__CLOUDBOOK:LOCAL__
def datos_locales(ruta):
    #Aquí se busca en la ruta y se devuelven los datos correctamente
```

```
#__CLOUDBOOK:LOCAL__
def load_data(ruta):
    global x_train
    global y_train
    global x_test
    global y_test
    x_train,y_train,x_test,y_test = datos_locales(ruta)

#__CLOUDBOOK:DU0__
def main():
    load_data("./datos")

#SKYNNET:BEGIN_MULTICLASS_ACC_LOSS
load_data("./datos")
_DATA_TRAIN_X = x_train
_DATA_TRAIN_Y = y_train
_DATA_TEST_X = x_test
_DATA_TEST_Y = y_test
```

Como los datos están en una ruta local, se usa una función que la mira y devuelve los datos correctos, la función *datos_locales(ruta)*

Como la función de carga de datos es local, los datos se declaran de manera global sin valor, y es en la función load_data donde se les asigna el valor. A esta función **hay que invocarla obligatoriamente en el main, y en la etiqueta skynnet** (esto es para que en el entorno distribuido, el agente 0, que puede no ejecutar funciones paralelas pueda tener los datos gracias a la invocación del main, y el resto de agentes que ejecutan funciones paralelas tengan todos los datos gracias a la invocación dentro de la etiqueta skynnet)

La función load_data acepta una ruta a los datos, con el ./ indicamos que está la carpeta en el directorio de trabajo actual, es decir en la ruta donde ejecutas el script de manera normal o en la carpeta */distributed/working_dir* cuando ejecutas en SkyNNet.

Por último, si quieres ejecutar el código de manera local y la carga de datos es una función costosa, por seguridad habría que hacer lo siguiente:

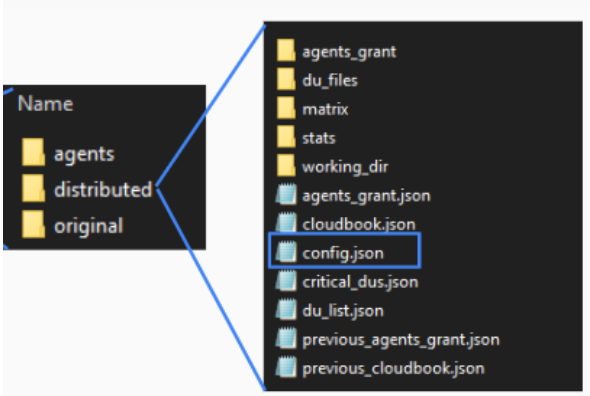
```
#SKYNNET:BEGIN_MULTICLASS_ACC_LOSS
if x_train == None:
    load_data("./datos")
_DATA_TRAIN_X = x_train
_DATA_TRAIN_Y = y_train
_DATA_TEST_X = x_test
_DATA_TEST_Y = y_test
```

2.2.2. Preparación para la ejecución en SkyNNet

Después de haber importado el entorno, instalado el repositorio y crear la carpeta en el working dir prepararemos para ejecutar Skynnet, es decir crearemos los proyectos etc…

1. generamos el código que queramos ejecutar en skynnet.
2. crearemos una carpeta llamada igual que el nombre del proyecto.
3. ejecutaremos el siguiente comando py cloudbook_maker.py -project_folder tu_proyecto.

   generando las siguientes carpetas.



4. La carpeta distributed que se ha generado con el comando anterior se tiene que compartir entre todas las máquinas que se quiera ejecutar el proyecto. Se comentará cómo se comparte esta carpeta en Anexo.

Información sobre cloudbook: La salida de los agentes en Skynnet se muestra por la misma consola en la que arrancas la gui, salvo la salida del agente 0, que se muestra en una nueva consola.

Información sobre SkyNNet: Si se quiere hacer la deconstrucción de la red directamente en el maker, hay que añadir la opción -skynnet al comando de ejecución del maker:

py cloudbook_maker.py -project_folder proyecto [-skynnet [n]]
De esta manera se preparará el código para distribuirse en n subredes, si no se introduce n se distribuirá en tantas subredes como número deseado de unidades desplegables haya en el fichero de configuración del proyecto

Para ejecutar los siguientes componentes se hace de la siguiente manera:
-Deployer: *py cloudbook_deployer -project_folder proyecto [-fast_start]*
-Launcher: *py cloudbook_run project_folder proyecto*
-Agente: *py gui.py* (intefaz gráfica con agentes y proyectos)

En el proyecto SkyNNet no se lanzarán los agentes directamente, si no que se hará mediante el dashboard del proyecto como se explica a continuación.

3. Uso de SKYNNET

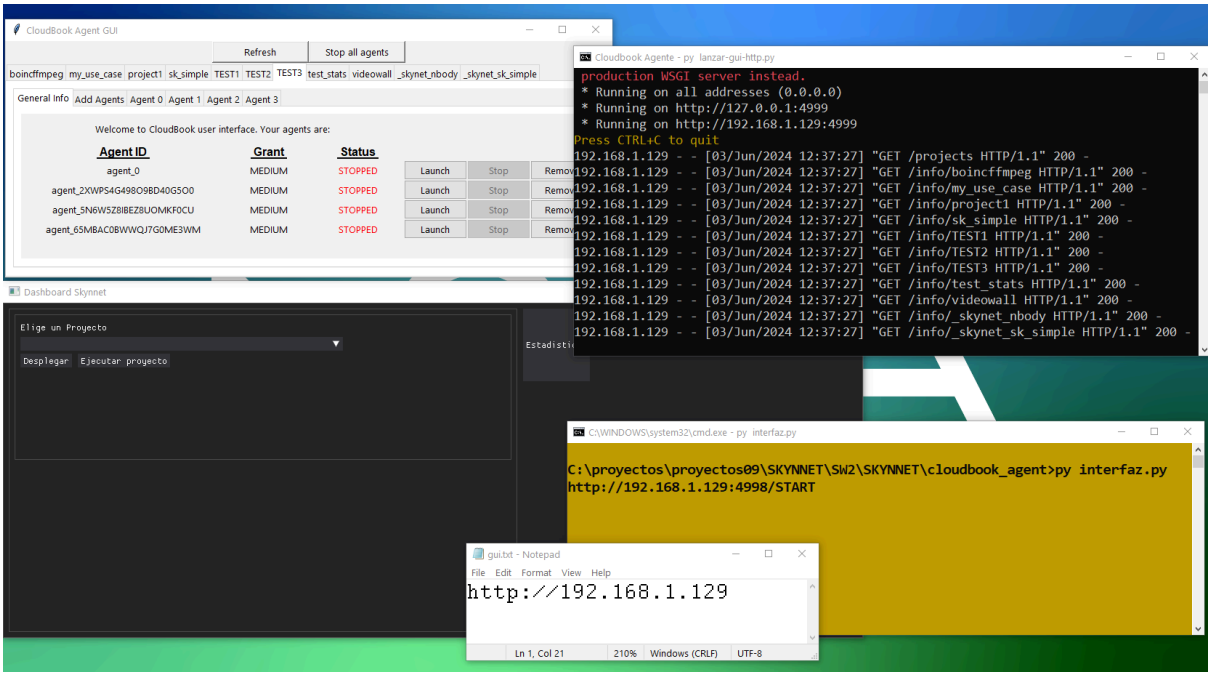
Para lanzar cualquier programa se deben dar dos pasos: primero se pasa a través de SK_tool cada programa de ejemplo y la salida se incorpora como fuente de entrada para el maker. Para facilitar la ejecución de los distintos comandos (maker, agentes, deployer, run) se hace uso del script que lanza las 4 consolas que necesitamos (**cb_start.bat**), el cual lanza 4 consolas en los directorios de las herramientas, una de cada color. este script se encuentra en el raiz del repositorio “skynnet” https://github.com/SKYNNET-ORG/SKYNNET

Después de compartir la carpeta "Distributed", se ejecutará el archivo interfaz.py, ubicado en la carpeta "agentes" dentro del repositorio.

El archivo interfaz.py es una interfaz grafica donde el usuario va a poder lanzar,detener y eliminar agentes, así como añadir nuevos agentes y ver estadísticas en tiempo real, para ejecutar este archivo, se deben seguir los siguientes pasos:

1. Ejecutar el archivo **lanzar-gui-http.py** en cada una de las máquinas (ubicado en la carpeta “SKYNNET/cloudbook_agent”. Este archivo se encargará de iniciar las interfaces gráficas (GUIs).
2. En la máquina del agente cero (desde donde hacemos el make) escribir las IPs de cada una de las máquinas en el archivo gui.txt, utilizando el siguiente formato: http://192.168.2.231. Esto es necesario para realizar las operaciones HTTP correspondientes.
3. Ejecutar el archivo interfaz.py. Este programa debe ejecutarse en la máquina que está ejecutando el agente 0, la cual se asume que es donde se encuentra el usuario que administra el proyecto. Se arranca automáticamente el gui del agente en cada máquina pero trabajaremos con el dashboard que se lanza automáticamente un poco después

ejemplo con una sola máquina (cuya IP acaba en 129):



Acciones relacionadas con los agentes :

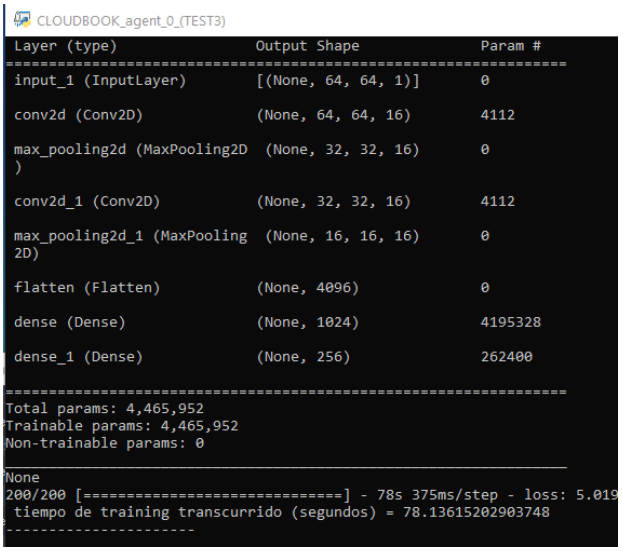
1. Seleccionar un proyecto en el menú desplegable, donde se listarán todos los proyectos existentes.
2. Esta acción desplegará una tabla que muestra todos los agentes y su información.

añadir

id	status	gui	grant	botons		
agent_0	parado	http://192.168.2.231	MEDIUM	lanzar	parar	eliminar
agent_QVTWINTAYT45SIAT7V72	parado	http://192.168.2.231	MEDIUM	lanzar	parar	eliminar

Para lanzar/parar/eliminar un agente se debe pulsar al botón correspondiente en la fila del agente que queramos realizar la acción.

al lanzar el agente cero se levantara una consola especifica de este agente (ya que se supone que el usuario se encuentra en la misma maquina donde se ejecuta el agente cero)

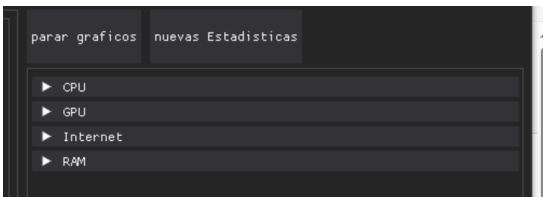


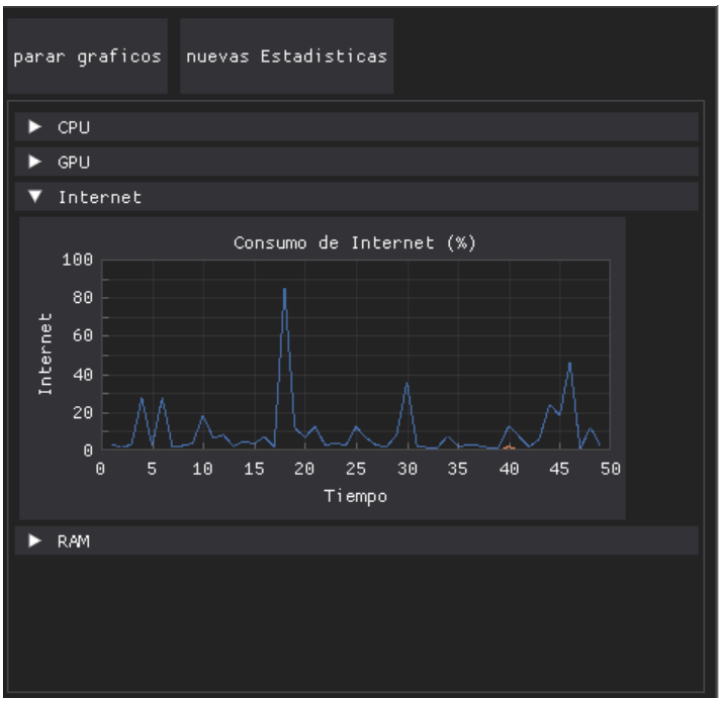
Para añadir un agente se debe pulsar el botón añadir encima de la tabla de agentes del proyecto. Este botón generará una ventana donde hay varios desplegables, el primer desplegable es para el GRANT el segundo para seleccionar en qué máquina se quiere crear el agente y pòr último hay un checkbox para indicar si es agente 0 o no lo es, al darle añadir se añadirá un agente a la tabla de agentes.

Ver estadísticas en tiempo de ejecución:

Para ver las estadísticas en tiempo de ejecución, se deben seguir los siguientes pasos:

1. se seleccionará un Proyecto
2. se seleccionara una máquina
3. dos opciones: la primera si se pulsa el botón estadísticas se verán las estadísticas de la máquina seleccionada. Y la segunda opción si se escoge un agente y se pulsa el botón de estadísticas se verán las estadísticas de ese agente. El botón estadísticas se encuentra en la parte derecha
4. Si se selecciona otro agente u otra máquina habrá que pulsar el botón de nuevas estadísticas.





3.1. Resumen de requisitos

- El archivo gui.txt tiene que estar en el directorio de trabajo, es decir en la misma carpeta que el archivo interfaz.py (dashboard)
- el archivo lanzar-gui-http.py tiene que estar en el mismo directorio que el archivo gui.py (la interfaz gráfica del agente de Skynnet)
- módulos de python que se necesitan para el dashboard.
    - request
    - psutil
    - dearpygui.dearpygui
    - threading
    - enum
- módulos de python que se necesitan para el servicio lanza-guis
    - flask
    - subprocess
    - werkzeug.serving
- requerimiento del gui.py
    - La carpeta distributed de cada proyecto tiene que estar compartida entre todas las máquinas y todas las especificaciones de cloudbook.

4. Anexo cómo compartir carpeta Distributed

En este anexo se explican tres formas de compartir la carpeta Distributed.

4.1. Ejecución con NFS FS (Sistema de Archivos de Red)

Suponga un escenario en el que desea una carpeta "distribuida" implementada usando NFS. Los pasos son los siguientes:

1. Un agente debe crear la carpeta compartida y comenzar a compartirla. Instalar el servidor NFS Kernel:

   sudo apt install nfs-kernel-server

2. Crear el directorio de exportación:

   sudo mkdir /jack/home/cloudbook/project/distributed

3. Asignar acceso del servidor a los clientes a través del archivo de exportación NFS: abrir etc/exports y escribir una línea para cada máquina que utilizará la carpeta:

   sudo nano /etc/exports

   La línea es:

   /jack/home/cloudbook/project/distributed clientIP(rw,sync,no_subtree_check)

4. Exportar el directorio compartido:

   sudo exportfs -a

5. Reiniciar el kernel:

   sudo systemctl restart nfs-kernel-server

6. Todos los usuarios deben:

   Instalar nfs-common:

   sudo apt-get install nfs-common

7. Crear un punto de montaje para la carpeta compartida del host NFS:

   sudo mkdir /jhon/home/cloudbook/project/distributed

8. Montar el directorio compartido en el cliente:

sudo mount serverIP:/jack/home/cloudbook/project/distributed /jhon/home/cloudbook/project/distributed

4.2. Ejecución con Google Drive FS

Suponga un escenario en el que desea una carpeta "distribuida" implementada usando Google Drive. Los pasos son los siguientes:

1. Crear una carpeta en Google Drive. Use el nombre que prefiera, por ejemplo "simple". Este archivo se utilizará (más adelante) como la "carpeta distribuida" de su proyecto.
2. Compartir su carpeta "simple" (creada en Google Drive) con otros usuarios o crear un enlace de uso compartido (con permisos de lectura/escritura).
3. Cada usuario debe mover la carpeta compartida "simple" a "Mi unidad".
4. Cada usuario debe instalar en su computadora la herramienta de sincronización de Google y sincronizar la carpeta "simple".
5. Crear un enlace simbólico a la carpeta de sincronización "simple" con el nombre "distributed". Este enlace debe crearse en la carpeta del proyecto cloudbook con el nombre "distributed". Use rutas absolutas para evitar problemas (de lo contrario, Windows crea el enlace en la carpeta system32).

   mklink /d "C:\Users\jack\cloudbook\simple\distributed" "C:\Users\jack\Google Drive\simple"

Tenga en cuenta que el tiempo de sincronización es largo (20 segundos o más).

4.3. Ejecución con Microsoft OneDrive FS

Suponga un escenario en el que desea una carpeta "distribuida" implementada usando Microsoft OneDrive. Los pasos son los siguientes:

1. Crear una carpeta en OneDrive y compartirla con otras personas, o crear un enlace compartible para enviar.
2. Para cada máquina, ingresar a la carpeta (vía web) y presionar "sincronizar". La carpeta ahora se está sincronizando.
3. Crear un enlace simbólico con mklink, de manera similar al ejemplo de Google Drive.