Anexo 3.1 Manual SK_TOOL

CONSORCIO:

NOKIA

ORGANISMOS DE INVESTIGACIÓN Y COLABORADORES:

Ciemat
Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas

SONRYE fotografía

red.es

Financiado por la Unión Europea
NextGenerationEU

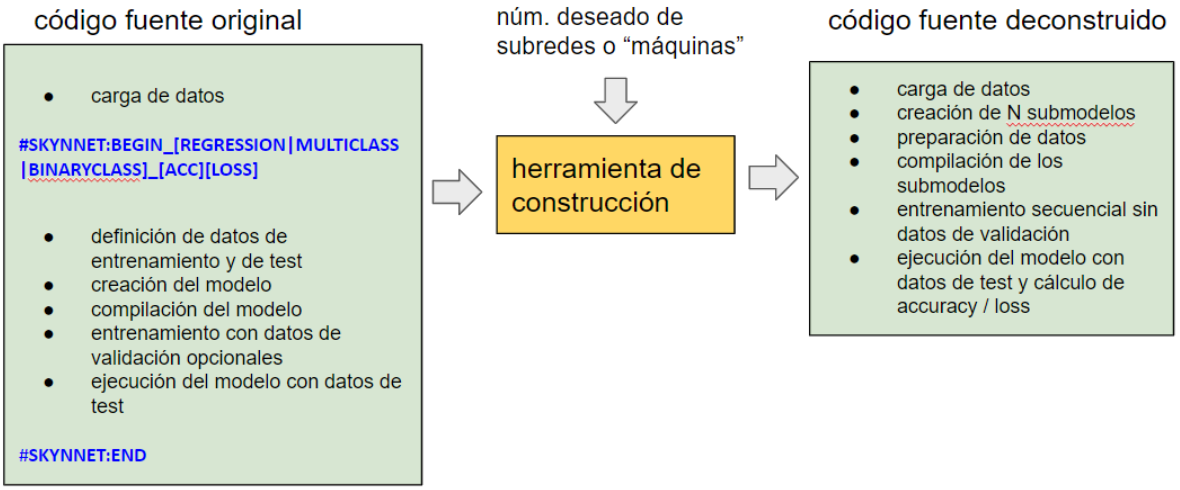
Plan de Recuperación, Transformación y Resiliencia

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

1. Contenido

1. Introducción

Este documento es el Manual de la herramienta sk_tool, para convertir el código de una red neuronal en código compatible con SkyNNet para su ejecución distribuida y en paralelo mediante etiquetado de código.



En este manual se recogen instrucciones sobre la instalación y uso de la herramienta, sobre la preparación del código fuente original, y sobre el funcionamiento del código fuente de salida que está preparado para funcionar tanto de manera local como distribuida y paralela.

Información: En el apartado 9 de este manual hay un resumen de requisitos y consejos de programación útil para tener en cuenta a la hora de hacer los programas que usen SkyNNet.

1.1. Instalación y uso

A continuación se detallan los pasos necesarios para instalar y usar la herramienta:

1. Descarga de la herramienta sk_tool: https://github.com/SKYNNET-ORG/sk_tool
2. Requisitos:
   - 2.1. Si usas python básico: Versión 1.0.1 de la librería ast_comments (descargar via pip >pip install ast-comments=1.0.1)
   - 2.2. Si usas Anaconda: El fichero del entorno se encuentra en la carpeta descargada *sk_tool.yaml*
3. En la carpeta descargada de git se encuentran los siguientes ficheros y carpetas:
   - 3.1. Ficheros:
     - sk_tool.py: Código fuente de la herramienta
     - sk_extra_codes.py: Código auxiliar de la herramienta
     - sk_tool.yaml: Entorno de anaconda
   - 3.2. Carpetas:
     - input: Ficheros de entrada de la herramienta
     - output: Ficheros de salida de la herramienta
       - Contiene un fichero llamado test_all.bat: Es un fichero para probar todos los scripts que haya en la carpeta output, útil para realizar pruebas
     - test: Batería de ficheros para ejecutar pruebas

   Dentro de las carpetas se encuentran ficheros de pruebas de concepto a modo de ejemplo.

4. Para uso normal:
   4.1. Colocas el fichero original en la carpeta inputs, por ejemplo *sk_tool/inputs/original.py*
   4.2. Ejecutas: >py sk_tool.py original numero_subredes_deseadas
   4.3. En la carpeta output se genera el fichero: *sk_tool/output/sk_original.py*
   4.4. Ejecutas la red deconstruida en *sk_tool/output/sk_original.py*
5. Para uso en modo test de uno o varios ficheros:
   5.1. Colocas los ficheros originales en la carpeta test
   5.2. Ejecutas: >py sk_tool.py test
   5.3. En la carpeta output deconstruye todos los ficheros que hay en la carpeta test para cuatro subredes.

1.1.1. Entorno Anaconda

La configuración de entorno de anaconda se encuentra en git, como sk_tool solo tiene una dependencia, también se han incluido las dependencias necesarias para ejecutar todas las pruebas de concepto y las pruebas masivas

1.2. Salida de la ejecución

Además del fichero con la red deconstruida, una vez ejecutas la herramienta se muestran unas trazas por pantalla que se explican a continuación:

```
#Fichero y numero de subredes deseadas
Fichero: mlp2.py en 4 subredes
mlp2.py
num subredes original 4

#Análisis de la etiqueta Skynnet
[{'Type': 'MULTICLASS', 'Options': '_ACC_LOSS'}]

#Información del análisis combinatorio, categorías iniciales y por
subred
Son 10 categorías originales
Para formar subredes=C(3, 2) es decir 3 categorias tomados de 2 en 2
número de grupos es 3
El número de subredes va a ser 3
Categorías por subred: 6.666666666666667
la reducción será por 1.5 esto es por defecto

#Información sobre el modelo analizado: Nombre, partes del modelo, no
tiene porque estar todas, solo las que haya en el código original
    name:model
    creation:<ast.Assign object at 0x000001A9346EEE00>
    summary:<ast.Call object at 0x000001A9346EF070>
    compile:<ast.Call object at 0x000001A9346EF760>
    fit:<ast.Call object at 0x000001A9346EF9A0>
```

```
    predict:<ast.Call object at 0x000001A934A5C250>
    data_train:[<ast.Assign object at 0x000001A9346A2230>, <ast.Assign
object at 0x000001A9346A2290>]
    data_val:[]
    data_test:[<ast.Assign object at 0x000001A9346A1300>, <ast.Assign
object at 0x000001A9346A0DC0>]
    input:<ast.Assign object at 0x000001A9346EE530>
```

1.3. Preparación de código fuente original

En este capítulo se explica cómo preparar el código fuente original para trabajar con la herramienta sk_tool

Nota sobre tiempos: Si se quiere obtener el tiempo de cada subred, se asume que también interesa el tiempo de la red sin deconstruir, por lo que se recomienda al usuario que la medida de tiempo la realice el en el modelo sin deconstruir, esta medida se replicará en los modelos deconstruidos.

Hay que tener en cuenta la carga de datos de entrenamiento, validación y test y el uso de etiquetas Skynnet para indicar el código a deconstruir. Las etiquetas son las siguientes, #SKYNNET:BEGIN y #SKYNNET:END que se ponen al principio y al final de la red.

Requisito obligatorio: 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. Explicación en el apartado “Función main y uso global” de este manual.

por ejemplo:

```
#__CLOUDBOOK:DU0__
def main():
    if hasattr(main,'executed'):
        return
    else:
        setattr(main,'executed',True)
```

Si no se usa una función main, la función vacía es así:

```
def main():
      pass
```

A la hora de usar la herramienta Skynnet, hay que tener en cuenta los aspectos referidos al código fuente que se describen en los siguientes apartados.

1.3.1. Imports y carga de datos

Esta parte del código se realiza fuera de las etiquetas Skynnet, por compatibilidad con el *maker* de Skynnet se introducen las siguientes restricciones (**Si no se cumplen estas restricciones el programa**

funciona de manera local, pero tendrá problemas cuando se ejecuta de manera distribuida y paralela en Skynnet):

- Los imports tienen que ser de un módulo por línea, y todos los imports juntos sin líneas de código entre medias
- Los datos no se pueden cargar en forma de tupla, si se tiene una función que devuelve una tupla se debe devolver cada elemento de la tupla en líneas de código separadas
- Antes de la carga de datos se debe incluir la etiqueta de Skynnet de ejecución distribuida #CLOUDBOOK:NONSHARED
- Si se realizan transformaciones sobre los datos deben ser operaciones sencillas, si es necesario hacer una operación compleja en una función externa, se recomienda que se haga en otro módulo o dentro de la etiqueta Skynnet, en este caso ten en cuenta que se va a realizar la función en cada subred.

```
#Esta es la forma original
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
#Normalización de datos
x_train = x_train / 255.0
x_test =  x_test / 255.0

#Esta es la forma correcta para sk_tool
#__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]

#otra opción
mnist_loaded = mnist.load_data
x_train = mnist_loaded[0][0] ...

#La normalización sigue igual ya que es una operación sencilla
x_train = x_train / 255.0
x_test =  x_test / 255.0

#SKYNNET:BEGIN
(Aquí comienza el código de la red)
```

Los casos en los que se quiere operar con los datos de forma compleja se tienen que simplificar de la siguiente manera:

```
#Forma original (filtro para hacer datos binarios, y adaptación
compleja)
y_train[y_train%2==0]=0
y_train[y_train%2!=0]=1
valid_samples=adaptImages(x_train,samples,data_type, a, h2,w2,
```

```
channels2)

#Forma simplificada
y_train = np.where(y_train % 2 == 0,0,1)
#Parte izquierda de la asignación sencilla y la operación se hace en la
derecha (además invoca a funciones externas de numpy)
valid_samples=loaderlocal.adaptImages(x_train,samples,data_type, a,
h2,w2, channels2)
#La función adaptImages está en otro módulo, otra opción es generar los
valid samples dentro de la etiqueta Skynnet
```

Datos propios en SkyNNET: En el caso de usar datos locales (sin descargar un dataset) tanto los datos, como las funciones para su carga (y opcionalmente su manipulación) deben ubicarse en la carpeta ***/distributed/working_dir*** del proyecto Skynnet. En el ejemplo anterior estaría el fichero *loaderlocal.py* con la función *adaptImages.* **Esta indicación es para skyNNet no para sk_tool**

1.3.2. Etiqueta Skynnet

Al inicio y final de la red se deben colocar las siguientes etiquetas:

```
#SKYNNET:BEGIN_[REGRESSION|MULTICLASS|BINARYCLASS]_[ACC][LOSS]
#Aquí va el código fuente
#SKYNNET:END
```

Las distintas opciones de la etiqueta begin sirven para:

- Tipo de problema que se quiere resolver: Regresión, clasificación multiclase, clasificación binaria.
- Medida de la red deconstruida que se quiere obtener: Accuracy, loss, ambas, o ninguna (ten en cuenta que estas medidas se hacen sobre el uso de la red, el predict compuesto)

1.3.3. Conjunto de variables

Tras la etiqueta begin se debe definir un conjunto de variables para la herramienta, de modo que se identifiquen con precisión los datos de entrada y de definición y uso de la red.

```
#SKYNNET:BEGIN_[REGRESSION|MULTICLASS|BINARYCLASS]_[ACC][LOSS]
_DATA_TRAIN_X = x_train
#Se usa x_train o el nombre en el que hayas cargado los datos
previamente
_DATA_TRAIN_Y = y_train
_DATA_TEST_X = x_test
_DATA_TEST_Y = y test
_NEURON_1 = 128
_NEURON_2 = 60
```

```
_NEURON_3 = 10
_EPOCHS = 10

<código fuente>

#SKYNNET:END
```

Las variables de datos, son las variables de entrenamiento, validación (si no usas un *validation_split* serían _DATA_VAL_X/Y) y test.

Para entrenar se van a usar variables de entrenamiento y de validación, y para ejecutar la red se usarán las variables de 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 (3 en el ejemplo) es la capa de salida, la herramienta identifica por defecto las siguientes variables:

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

Embedding se usa en transformers, cell en algunas recurrentes o convolucionales, al igual que filters, de todas formas, **el uso de estas variables es para indicar a la herramienta que capas tiene que reducir al deconstruir**, y se han tomado a partir de las reducciones propuesta en el documento E2.1.

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

1.3.4. 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, como se ve en el siguiente ejemplo:

Como se ve, las variables que se van a reducir son las épocas y las capas ocultas, el shape de la capa de entrada no se reduce y se deja igual, como ya se ha visto en el documento E2.1.

```
#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
_NEURON_1 = 128
_NEURON_2 = 60
_NEURON_3 = 10
_EPOCHS = 10

#Modelo funcional
inputs = tf.keras.Input(shape=(28,28))
x = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(_NEURON_1, activation='relu')(x)
```

```
x = tf.keras.layers.Dense(_NEURON_2, activation='relu')(x)
outputs =  tf.keras.layers.Dense(_NEURON_3, activation='softmax')(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)

print(model.summary())

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(_DATA_TRAIN_X, _DATA_TRAIN_Y, validation_split=0.3,
epochs=_EPOCHS)
predicted = model.predict(_DATA_TEST_X)
#SKYNNET:END
```

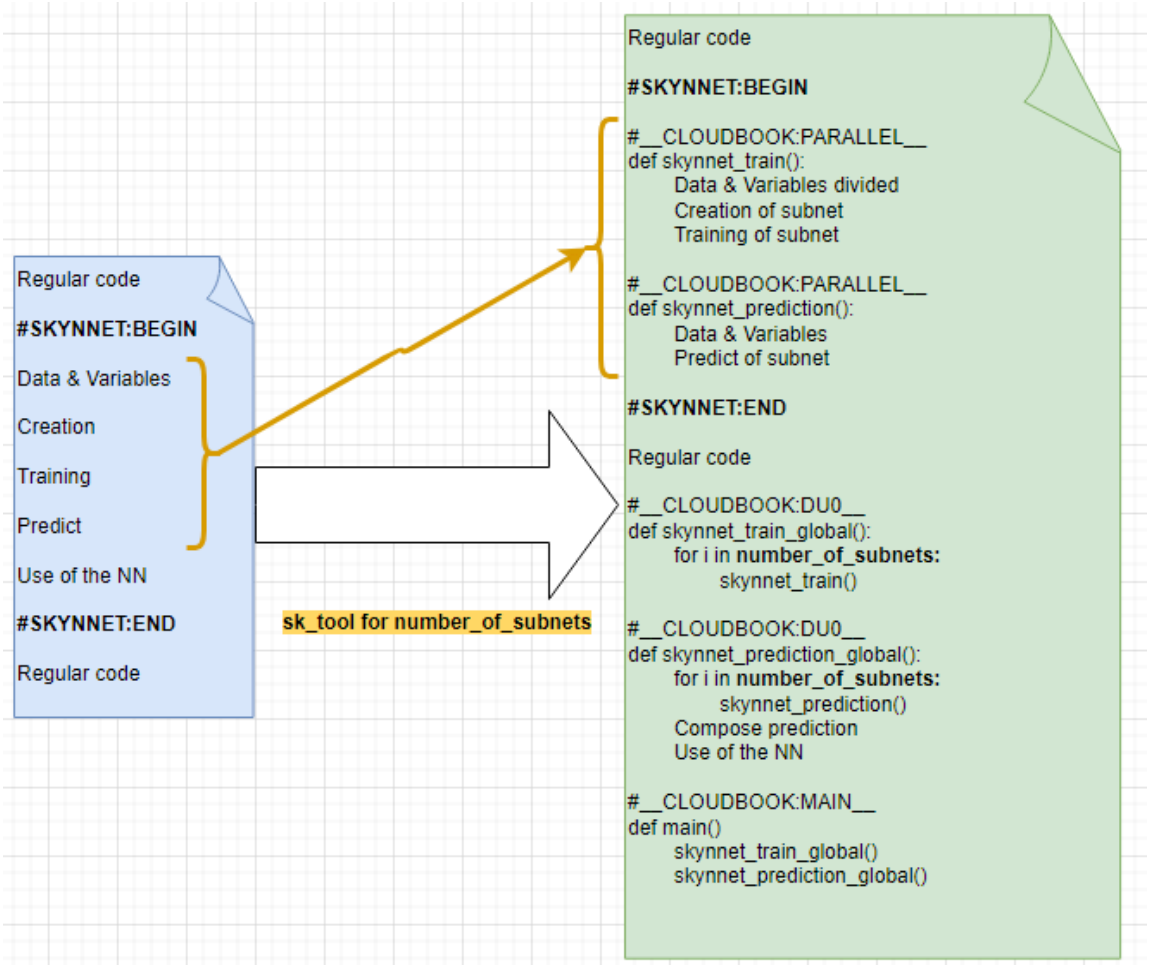
En la función fit se observa que se usan las variables _DATA_TRAIN, no las que se cargan al inicio (en el ejemplo x_train para las variables e y_train para las etiquetas)

Estos parámetros que se indican al principio, son parámetros genéricos que serán reducidos en las subredes para deconstruir la red.

1.4. Explicación código de salida

A continuación se explica como funciona el código de salida con la red deconstruida, hay que tener en cuenta que el código original se divide en tres secciones:

- **Entrenamiento y predict de la red original:** Se generan dos funciones para entrenar y usar las subredes generadas
- **Entrenamiento y predict global:** Se generan dos funciones que invocan a las funciones anteriores tantas veces como subredes sean necesarias. En caso de que se use, es en estas funciones donde se compone la predicción de cada subred.
- **Punto de entrada del programa:** Por defecto se crea un punto de entrada para el programa, ya que originalmente el uso estaba en el código a nivel de módulo (sin indentación) y para ejecutarlo de manera distribuida y paralela en skynnet es necesario indicar cuál es el punto de entrada o *main* del programa.



1.4.1. Variables globales

Las variables globales en modo local funcionan de la misma manera, pero si se usan en skynnet hay dos tipos distintos:

- Variables globales: Son únicas para todo el programa distribuido.
- Variables nonshared: Son únicas para cada ordenador de la red distribuida.

Además de la carga de datos que se indicó que era nonshared en el apartado 8.2 tenemos las siguientes:

```
#__CLOUDBOOK:GLOBAL__
predictions_0 = {}
#Diccionario que contendrá las predicciones de los submodelos
#__CLOUDBOOK:NONSHARED__
autoencoder = [None, None, None, None]
#Lista de submodelos, uno por cada subred, el modelo original se
convierte en lista de submodelos. Cada máquina tiene distinto de “None”
los modelos que haya creado.
to_predict_models = []
#Lista de submodelos entrenados para indicar que se pueden usar (hacer
predict), esto es útil en un entorno distribuido. Cada máquina solo hace
predict de los modelos que haya entrenado
```

1.4.2. Entrenamiento de la red original

Aquí se traduce la mayor el entrenamiento de la red original, se produce la función **skynnet_train_X**, donde X es el índice de red del código original (puede haber varios bloques de etiquetas #Skynnet:Begin/End, se numeran de 0 a n-1)

Esta función contiene la definición, compilación y entrenamiento de la red original (no el predict o su uso posterior), pero con los datos tanto de entrenamiento y validación como la composición de la red reducidos para adaptarse al tamaño de la subred que sea necesario.

Recibe un parámetro que es el índice (sk_i=skynnet index) de subred, de 0 a n-1, siendo n el número de subredes, y permitiendo así identificar a cada subred que entrenará distintos datos, o producirá una salida distinta.

En esta función se van a reducir los datos según la deconstrucción aplicada (como se explica en el documento E.2.1 y se analiza en el E.4.4), se va a generar código para que en tiempo de ejecución se calcule con qué datos se va a entrenar (o validar) la subred, y se va a definir, compilar y entrenar la subred con los parámetros genéricos de los datos deconstruidos.

1.4.2.1. Reducción de parámetros genéricos y preparación de variables globales

La primera parte de la función es la carga de variables globales (ya que se van a modificar, la lista de modelos que antes eran None ahora tendrá un modelo compilado, y habrá un modelo listo para hacer predict en to_predict_models)

```
#CÓDIGO ORIGINAL
#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
_NEURON_1 = 128
```

```
_NEURON_2 = 60
_NEURON_3 = 10
_EPOCHS = 10

#CÓDIGO DECONSTRUIDO: En este caso con un factor de deconstrucción de
1.5
#__CLOUDBOOK:PARALLEL__
def skynnet_train_0(sk_i):
      global model
      global to_predict_models
      _DATA_TRAIN_X = x_train
      _DATA_TRAIN_Y = y_train
      _DATA_TEST_X = x_test
      _DATA_TEST_Y = y_test
      _NEURON_1 = 86
      _NEURON_2 = 40
      _NEURON_3 = 7
      _EPOCHS = 7
```

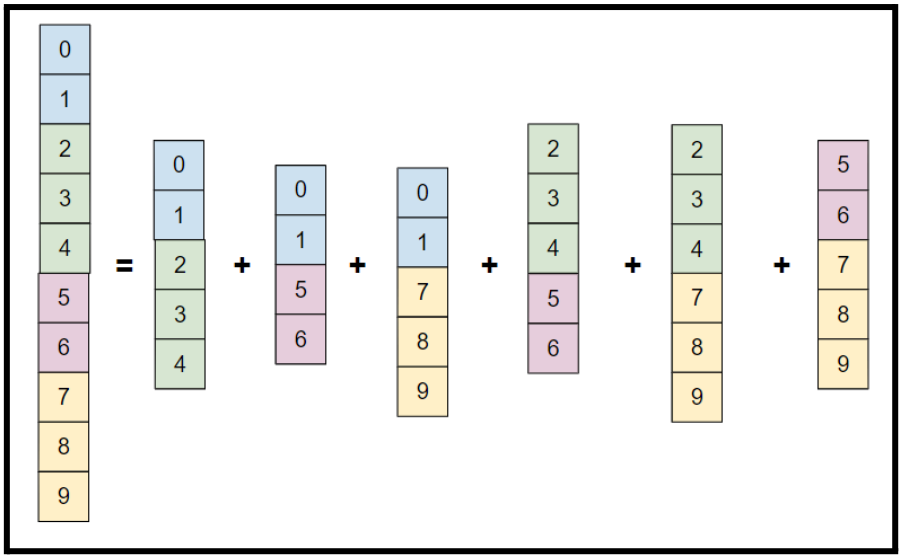
1.4.2.2. Selección de datos de entrenamiento de la subred

Antes de generar la red, hay que preparar el código para que en tiempo de ejecución cada subred sepa con qué datos trabajar. Esta tarea es distinta según el tipo de problema que se trate. Como se ve en los documentos E.2.1 y E.4.4.

Para problemas de clasificación multiclase: Los datos se dividen según la fórmula

$$number\ of\ subnets\ = \frac{\left(\frac{N}{(n/2)}\right)!}{2\left(\frac{N}{(n/2)}-2\right)!}$$

Para ello se van a escribir 2 funciones auxiliares que realizan el cálculo combinatorio del número de subredes, y dividen las categorías originales en los grupos necesarios para dividir los datos y que todos se comparen con todos. A continuación se ve un ejemplo para entenderlo de forma sencilla



La función *dividir_array_categorias* es la que asigna los colores a las categorías iniciales y la función *combinar_arrays* es la que combina los colores en las n subredes finales

El código queda como se muestra a continuación:

```
grupos_de_categorias = dividir_array_categorias(_DATA_TRAIN_Y, 10, 3)
combinacion_arrays = combinar_arrays(grupos_de_categorias)[sk_i]

#Una vez conocidas las categorías, se filtran los datos
_DATA_TRAIN_X = _DATA_TRAIN_X[np.isin(_DATA_TRAIN_Y,
combinacion_arrays)]
_DATA_TRAIN_Y = _DATA_TRAIN_Y[np.isin(_DATA_TRAIN_Y,
combinacion_arrays)]

#Se introduce un comentario de información de Skynnet sobre la división
de datos
print('====================================')
print('Skynnet Info: Longitud de los datos de la subred
(datos,etiquetas):', len(_DATA_TRAIN_X), len(_DATA_TRAIN_Y))
print('Skynnet Info: Categorías de esta subred',
np.unique(_DATA_TRAIN_Y))
print('====================================')

#Se convierten las etiquetas a categorías consecutivas desde 0 para
trabajar con tensorflow
categorias_incluir = np.unique(_DATA_TRAIN_Y)
etiquetas_consecutivas = np.arange(len(categorias_incluir))
_DATA_TRAIN_Y = np.searchsorted(categorias_incluir, _DATA_TRAIN_Y)

#Se corrige la capa de salida en los casos que por combinatoria no todos
los modelos clasifiquen el mismo número de categorías
_NEURON_3 = len(np.unique(_DATA_TRAIN_Y))
```

Para problemas de clasificación binaria: En este caso se reducen los datos a la mitad

```
datos_train_x_1 = _DATA_TRAIN_X[:len(_DATA_TRAIN_X) // 2]
datos_train_x_2 = _DATA_TRAIN_X[len(_DATA_TRAIN_X) // 2:]
datos_train_y_1 = _DATA_TRAIN_Y[:len(_DATA_TRAIN_Y) // 2]
datos_train_y_2 = _DATA_TRAIN_Y[len(_DATA_TRAIN_Y) // 2:]
if sk_i == 1:
      _DATA_TRAIN_X = datos_train_x_1
      _DATA_TRAIN_Y = datos_train_y_1
else:
      _DATA_TRAIN_X = datos_train_x_2
      _DATA_TRAIN_Y = datos_train_y_2
_NEURON_3 = 2 #Por defecto la última capa es 2
```

Para problemas de regresión: Se considera la estrategia de problemas sin localidad (ver documento E.4.4) y no se dividen los datos de entrada, solo se dividen los datos de salida en porciones

```
#Is not necessary to divide data
train_splits = np.array_split(_DATA_TRAIN_Y, 4, axis=1)
_DATA_TRAIN_Y = train_splits[sk_i]
#El tam de la ultima dimension
_NEURON_3 = _DATA_TRAIN_Y.shape[-1]
```

Por simplicidad se ha mostrado la división de los datos de entrenamiento, en caso de haber datos de validación la estrategia es la misma.

1.4.2.3. Definición y entrenamiento de la subred

Se traducen las funciones referidas al modelo, y la predicción de la red, no se tiene en cuenta porque va en otra función, su uso posterior tampoco se tiene en cuenta, porque irá en otra función distinta.

Código original:

```
inputs = tf.keras.Input(shape=(28,28))
x = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(_NEURON_1, activation='relu')(x)
x = tf.keras.layers.Dense(_NEURON_2, activation='relu')(x)
outputs =  tf.keras.layers.Dense(_NEURON_3, activation='softmax')(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)

print(model.summary())
start=time.time()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
```

```
              metrics=['accuracy'])


model.fit(_DATA_TRAIN_X, _DATA_TRAIN_Y, validation_split=0.3,
epochs=_EPOCHS)
end=time.time()
print (" tiempo de training transcurrido (segundos) =", (end-start))

predicted = model.predict(_DATA_TEST_X)

#SKYNNET:END
```

Código deconstruido:

```
inputs = tf.keras.Input(shape=(28, 28))
x = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(_NEURON_1, activation='relu')(x)
x = tf.keras.layers.Dense(_NEURON_2, activation='relu')(x)
outputs = tf.keras.layers.Dense(_NEURON_3, activation='softmax')(x)
model[sk_i] = tf.keras.Model(inputs=inputs, outputs=outputs)
print(model[sk_i].summary())
start = time.time()
model[sk_i].compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model[sk_i].fit(_DATA_TRAIN_X, _DATA_TRAIN_Y, validation_split=0.3,
epochs=_EPOCHS)
end = time.time()
print(' tiempo de training transcurrido (segundos) =', end - start)
to_predict_models.append(sk_i)
```

El código deconstruido conserva la medida de tiempo original, y añade la modificación de la lista to_predict_models.

1.4.3. Predict de la red original

Al analizar la red original, se trata de manera especial el predict y su uso posterior, puesto que se van a realizar en dos funciones distintas, el predict de la subred se va a hacer en la función **skynnet_prediction_X** donde X es el índice de la red, y el uso se realizará en la función global que se verá más adelante que compone las predicciones de cada subred.

En la función skyynet_prediction_X se va a realizar la predicción de la subred y se va a adaptar para poder combinarse con el resto de subpredicciones. Además genera código para su uso compatible con la ejecución en el sistema Skynnet de manera distribuida y paralela.

A continuación se explican estas características por separado, y al final del apartado se muestra una función completa a modo de ejemplo

1.4.3.1. Variables globales, carga de datos y predicción

Aunque el código original sea solo la instrucción del predict, en esta función hay que tener en cuenta los datos de test y las variables globales que hay que tratar para poder ejecutarlo de forma paralela en skynnet.

```
#CÓDIGO ORIGINAL
predicted = model.predict(_DATA_TEST_X)

#CÓDIGO DECONSTRUIDO
#__CLOUDBOOK:PARALLEL__
def skynnet_prediction_0():
      global predictions_0 #Diccionario de predicciones
      global to_predict_models #Lista de modelos entrenados en cada
máquina
      global model #Lista de modelos
      #Se necesitan los datos de test, que no se dividen al hacer
predict
      _DATA_TEST_X = x_test
      _DATA_TEST_Y = y_test

      predicted = model[sk_i].predict(_DATA_TEST_X, verbose=1)
```

1.4.3.2. Adaptación de la predicción y serialización

La predicción se tiene que adaptar, para luego juntar todas las predicciones de las subredes, y cada tipo de problema tiene una estrategia distinta:

Para clasificación multiclase: Las subredes clasifican menos categorías de las totales, por lo que se tienen que calcular qué categorías clasifica cada subred, y las categorías que no tenga se les asigna un valor por defecto que se sobreescribirá cuando se junten las predicciones

```
predicted = model[sk_i].predict(_DATA_TEST_X, verbose=1)
for (i, pred) in enumerate(predicted):
      array_final = np.ones(10) #Array de unos
      array_final[categorías] = pred
      #Las categorías de la subred usan los valores de la predicción,
      el resto se quedan a uno
      resul.append(array_final.tolist())
predictions_0[label] = resul
#Se actualiza la variable global con la predicción preparada, la
variable label es para saber quien es el dueño de la predicción,
contiene información de las categorías que clasifica y del índice de
submodelo (se verá más adelante)
```

Para clasificación binaria: No hace falta modificar porque las categorías son las mismas que en la red original

Para regresión: En los problemas de regresión no hace falta modificar porque cada sub red trabaja con toda la entrada y devuelve una porción de la salida, para el predict esto es igual.

La predicción de cada subred también se serializa para poder transmitirla sobre la red cuando se use en SkyNNet

1.4.3.3. Adaptación a ejecución en Skynnet

Para que se ejecute en skynnet hay que tener en cuenta que las predicciones se usarán en distintos ordenadores y el código resultante tiene que ser robusto para su uso distribuido, por lo que se usan características de Skynnet para la ejecución distribuida y paralela:

- La variable cloudbook: Esta es una variable especial que identifica a cada máquina que participa de la ejecución del programa (también se refiere a cada una como agente). Para saber a qué máquina pertenece cada predicción, se usa el identificador de agente como clave del diccionario *predictions_0*.

```
label = __CLOUDBOOK__['agent']['id']
[...]
predictions_0[label] = resul
```

Esto genera un problema ya que la variable cloudbook en una ejecución local no existe y produce un error en tiempo de ejecución. Este error se arregla con las etiquetas Beginremove y Endremove, que incluyen código que solo se tiene en cuenta en la ejecución local del programa.

```
#__CLOUDBOOK:BEGINREMOVE__
__CLOUDBOOK__ = {}
__CLOUDBOOK__['agent'] = {}
__CLOUDBOOK__['agent']['id'] = 'agente_skynnet'
#__CLOUDBOOK:ENDREMOVE__
#Ahora en modo local se puede usar una variable cloudbook por defecto,
para que no se llamen todas igual se le añade el índice de modelo con un
separador
label = __CLOUDBOOK__['agent']['id']+ ('_' + str(sk_i))
[...]
predictions_0[label] = resul
```

- Bucle de predicción sobre lista de modelos: Esta característica no es propia de Skynnet si no por el hecho de ejecutarse sobre la red de forma paralela y distribuida.

  Por ejemplo, si tienes dos máquinas pero tres subredes, ya sea por una limitación o porque una máquina haya fallado una de las dos máquinas tendrá que entrenar y ejecutar dos modelos.

  La máquina A tiene 1 modelo
  La máquina B tiene 2 modelos
  Se invoca la función predict 3 veces porque hay 3 submodelos
  - se invoca primero a la máquina A esta predice su modelo,
  - luego se invoca a la maquina b y predice uno de sus modelos,
  - luego se invoca a la máquina A de nuevo y no tiene el modelo que se quiere predecir, esto provoca un fallo.

Por eso la función predict, predice sobre los modelos que posee, si se le vuelve a invocar para predecir un modelo que ya ha predicho, no hace nada y continúa con la ejecución.

- Sección crítica: Para que dos invocaciones a un mismo agente no predigan los mismo modelos, se mete el bucle de predicción en una sección crítica con las etiquetas lock y unlock, de esta manera no se producen problemas por el acceso de distintos hilos a la predicción

El resultado de la función de predicción completa, es como se muestra a continuación (para un problema de clasificación multiclase):

```
#__CLOUDBOOK:PARALLEL__
def skynnet_prediction_0():
      global predictions_0
      global to_predict_models
      global model
      _DATA_TEST_X = x_test
      _DATA_TEST_Y = y_test
      #__CLOUDBOOK:BEGINREMOVE__
      __CLOUDBOOK__ = {}
      __CLOUDBOOK__['agent'] = {}
      __CLOUDBOOK__['agent']['id'] = 'agente_skynnet'
      #__CLOUDBOOK:ENDREMOVE__
      #__CLOUDBOOK:LOCK__
      for sk_i in to_predict_models[:]:
            to_predict_models.remove(sk_i)
            label = __CLOUDBOOK__['agent']['id'] + ('_' + str(sk_i))

grupos_de_categorias=dividir_array_categorias(_DATA_TEST_Y,10,3)
            categorias_incluir =
combinar_arrays(grupos_de_categorias)[sk_i]
            label += f'{categorias_incluir}'
            #No se dividen los datos, pero se calculan las categorías de
cada
            subred
            predicted = model[sk_i].predict(_DATA_TEST_X, verbose=1)

            #A continuación se extraen las categorías de la subred para
saber
            asignarlas correctamente al "estirar" la predicción
            categorias_str = label[label.find('[') + 1:label.find(']')]
            categorias = np.fromstring(categorias_str, dtype=int, sep='
')
            resul = []
            for (i, pred) in enumerate(predicted):
                  array_final = np.ones(10)
                  array_final[categorias] = pred
```

```
                resul.append(array_final.tolist())
            predictions_0[label] = resul
      #__CLOUDBOOK:UNLOCK__
```

1.4.4. Función main y uso global

A continuación se muestra el punto de entrada generado y cómo invoca a las funciones globales, en este ejemplo el número de subredes es tres.

Indicación sobre Cloudbook: En este punto no es necesario ser un experto en cloudbook, pero se generarán funciones de entrenamiento y uso de cada modelo, de manera que cuando se porte a cloudbook se puedan entrenar y ejecutar las subredes en paralelo, por ello se encontrarán etiquetas en algunas funciones, como *PARALLEL, LOCAL, DU0, SYNC.*
De todas formas se puede ejecutar el modelo en local (secuencialmente en lugar de en paralelo).

```
#__CLOUDBOOK:DU0__
def skynnet_train_global_0():
      for i in range(3):
            skynnet_train_0(i) #Entrenamiento de la subred
      #__CLOUDBOOK:SYNC__
#__CLOUDBOOK:DU0__
def skynnet_prediction_global_0():
      for i in range(3):
            skynnet_prediction_0() #Predict de la subred
      #__CLOUDBOOK:SYNC__

#__CLOUDBOOK:MAIN__
def sk_main():
      try:
          main()
      except:
          print("Skynnet Info: There is no original main")
      pass
      #El 0 es porque solo hay un bloque Skynnet
      skynnet_train_global_0()
      skynnet_prediction_global_0()

if __name__ == '__main__':
      main()
```

En la función prediction global es donde se compone la predicción de las subredes, así que puede incluir el código de lo que se hace con el predict o el cálculo de las medidas (acc y loss). **El uso original de la red se mantiene o se amplía con medidas extra.**

```
#__CLOUDBOOK:DU0__
```

```
def skynnet_prediction_global_0():
	_DATA_TEST_X = x_test
	_DATA_TEST_Y = x_test_out
	for i in range(4):
		skynnet_prediction_0()
	#__CLOUDBOOK:SYNC__
	global predictions_0
	predictions_0 = dict(sorted(predictions_0.items(), key=lambda x:
int(x[0].split('_')[-1])))
	reconstructed_img = np.concatenate(list(predictions_0.values()),
axis=1)
	mse = tf.keras.losses.MeanSquaredError()
	mse_orig = mse(y_test, reconstructed_img).numpy()
	print('==========================================')
	print('Skynnet Info: La loss compuesta es: ', mse_orig)
	print('==========================================')
	n = 20
	plt.figure(figsize=(20, 4))
	for i in range(n):
		ax = plt.subplot(2, n, i + 1)
		plt.imshow(_DATA_TEST_X[i].reshape(28, 28))
		plt.gray()
		ax.get_xaxis().set_visible(False)
		ax.get_yaxis().set_visible(False)
		ax = plt.subplot(2, n, i + 1 + n)
		plt.imshow(reconstructed_img[i].reshape(28, 28))
		plt.gray()
		ax.get_xaxis().set_visible(False)
		ax.get_yaxis().set_visible(False)
	plt.show()
```

1.4.5. Predicción compuesta

Como se ha visto en el apartado anterior la predicción compuesta se compone en la función con el predict global. El cálculo de la predicción compuesta se realiza como se explica en los documentos E.2.2 y E.4.4 y se indica a continuación:

- Clasificación multiclase: Se hace multiplicando las predicciones de las subredes.
- Clasificación binaria: Eligiendo por certeza entre las predicciones de cada subred.
- Regresión: Se concatenan la predicciones de forma ordenada (se tiene el índice en la etiqueta de predictions_x) ya que la salida se ha dividido hay que reconstruirla de manera ordenada.

La predicción compuesta permite:

- El uso de la red posterior al *predict*, por ejemplo reconstruir una imagen en un *autoencoder*
- El cálculo de la accuracy compuesta.

- El cálculo de la loss compuesta:
  - Problemas clasificación: En función de los datos de salida se usará SparseCategoricalCrossEntropy (cuando la salida tiene una dimensión) o CategoricalCrossEntropy (cuando la salida tiene más de una dimensión)
  - Problemas regresión: La función Mean Squared Error

1.5. Código de entrada avanzado: Funciones y clases para la red neuronal

En muchos casos el código de la red puede ser más complejo, requiriendo el uso de clases auxiliares o de funciones para generar subredes, o conjuntos de capas, en estos casos para preparar el código de entrada hay que hacer lo siguiente:

- Las clases auxiliares no hace falta tocarlas
- Las funciones auxiliares para crear la red neuronal se deben meter dentro del bloque skynnet, es decir, entre las etiquetas Skynnet:begin y end, y después de la definición de datos y variables de la red: Esto se hace, para que en el código de salida puedan acceder a los datos de entrenamiento, validación y test (en el código de salida la parte entre las etiquetas está en funciones, ya no se puede acceder a los datos como si fuesen variables globales).

- 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.

1.5.1. Función main propia

En caso que el código original necesite un punto de entrada del programa, es decir, una función main, debe cumplir con los siguientes requisitos para ser compatible con el código producido por esta herramienta:

- Tiene que llamarse main y no tener parámetros
- Tiene que estar fuera de las etiquetas Skynnet
- Etiquetada como cloudbook:du0, para que en la red de ordenadores distribuidos de skynnet se ubique donde está el punto de entrada del programa.
- Para ejecutarse una única vez (ya que el código que produce la herramienta tiene prevista su invocación, además de la invocación original) hay que incluir las siguientes líneas de código:

```
#__CLOUDBOOK:DU0__
def main():
    if hasattr(main,'executed'):
        return
    else:
        setattr(main,'executed',True)
```

Requisito obligatorio: 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

Si no se usa una función main, la función vacía es así:

```
def main():
      pass
```

1.5.2. Carga de datos en una función

La manera más sencilla de cargar los datos es realizar la carga a nivel de módulo, como se ha mostrado en los ejemplos, pero se puede querer cargar los datos en funciones propias, para realizar esto, se aconseja la siguiente manera:

- Los datos, ya sean de entrenamiento, validación y test tienen que ir en conjunto, ya que a las funciones *fit* y *predict* se invoca de forma automática con código generado por sk_tool.
- La invocación a estas funciones debe producirse de la siguiente manera:
    - En la función main propia: Al incluir esta invocación, tienes que invocar a la función main en el código original
    - Dentro de la etiqueta Skynnet:begin/end
    - Las variables de los datos se tienen que declarar como None

```
#__CLOUDBOOK:NONSHARED__
mnist = None
x_train = None
y_train = None

#__CLOUDBOOK:LOCAL__
def load_data():
      global mnist
      global x_train
      global y_train
      mnist=mnist.load()
      x_train = mnist[0]
      y_train = mnist[1]

#__CLOUDBOOK:DU0__
def main():
    if hasattr(main,'executed'):
        return
    else:
        setattr(main,'executed',True)
    load_data()
```

```
main()

#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
...
```

La motivación de esa estrategia, es que el uso de estas variables este preparado para la ejecución paralela y distribuida en Skynnet

- Se declaran como None, para que todos los ordenadores de la red distribuida tengan esas variables por defecto
- En la función load_data se cargan los datos, de manera que afecten a las variables globales. Se etiqueta como Cloudbook:Local para que la función esté en todos los ordenadores de la red distribuida.
- Se invoca dos veces, para asegurar que todos los ordenadores de la red tienen estos datos, ya que la función main local, puede ir a un ordenador distinto de donde irán las funciones paralelas que incluyen el código entre las etiquetas Skynnet.

Hay casos en los que conviene usar **otra estrategia**, esta es asignar a las variables de datos funciones auxiliares con la carga de datos, en lugar de depender de una función general para la carga de datos.

Este caso se da cuando se usan computadores de muy bajas prestaciones (o microcomputadores como una raspberry pi) que necesitan cargar los datos de test, solo cuando se van a usar, una vez terminado el entrenamiento de las subredes, para ahorrar recursos computacionales. En este caso la estrategia recomendada es la siguiente:

```
_DATA_TEST_X = load_data_test_x()
_DATA_TEST_Y = load_data_test_y()
```

Las funciones de carga de datos han de estar fuera de la etiqueta Skynnet, y recomendablemente en un fichero auxiliar, aunque esto no es un requisito obligatorio. Deben devolver los datos en un array compatible con numpy, ya que se analiza de forma automática características de los mismos como el *array.shape*.

1.5.3. Uso de generadores

Una opción avanzada de tensor flow es el uso de funciones generadoras , para generar los datos en pequeños conjuntos (*batches*) en lugar de tener siempre en memoria el conjunto completo. La herramienta sk_tool permite el uso de funciones generadoras.

Una función generadora en Python se diferencia de una función convencional porque, tras ejecutar la instrucción yield, devuelve el control al programa que la invocó. Sin embargo, la función no finaliza;

en su lugar, se pausa y guarda su estado actual (incluyendo los valores de las variables). Esto permite que la ejecución pueda reanudarse posteriormente desde donde se dejó.
Para definir un generador, se invoca la función generadora. Al iterar sobre el generador, se procesa y entrega el primer dato. El generador procesa un dato y se detiene en la primera instrucción yield que encuentra, devolviendo ese dato procesado.
En el contexto de TensorFlow, al utilizar un generador con imágenes, podemos devolver en cada yield un conjunto reducido de imágenes, es decir, las imágenes que componen el batch en cada iteración. De este modo, solo leemos las imágenes necesarias para cada batch, manteniendo en la memoria RAM únicamente las imágenes del batch actual. Esto permite un uso eficiente de la RAM.

La herramienta sk_tool permite el uso de funciones generadoras como sigue:

- **Requisito:** Para tener en cuenta los datos, la variable que hay que usar es _DATA_TRAIN_GEN_Y, que contendrá sólo una lista de categorías de la red neuronal, por ejemplo:

```
_DATA_TRAIN_GEN_Y = np.arange(0,numero_de_clases)
```

- La variable _DATA_TRAIN_GEN_Y en tiempo de ejecución una vez deconstruida, contendrá la lista de etiquetas correspondiente a cada subred.
- **Requisito:** La función fit de tensor flow es una función de orden superior, admite funciones como parámetros y esto no es compatible con el uso de SkyNNet en la red, por lo que la función generadora debe ubicarse en un fichero auxiliar.
- **Requisitos de la función generadora:**
    - La función generadora lee los datos y los procesa en un conjunto mas pequeño que devolverá al hacer yield
    - Parámetros de entrada:
        - batch_size (int): El tamaño del lote que se generará en cada iteración. Es el número de imágenes y etiquetas que se procesarán juntas en una única pasada por la red neuronal.
          combinacion_arrays (lista de int): Lista que contiene las categorías (como enteros) de interés. Solo se procesarán las imágenes que pertenezcan a estas categorías
    - Parámetros de salida:
        - Salida del generador (tupla de np.array): La función utiliza la declaración yield para devolver un lote de datos en cada iteración.
            - train_X (np.array de np.array de float): Un array de numpy de datos
            - train_Y (np.array de int): Un array de numpy de etiquetas que representan las categorías de las imágenes en el lote. Las etiquetas se convierten a índices que corresponden a sus posiciones en combinacion_arrays
- **Requisito de invocación de la función generadora:** La variable *steps_per_epoch* que tensor flow calcula directamente en este caso se la tiene que proporcionar el programador, ya que al usar un generador, tensor flow no sabe cuántos batches usará.

A continuación se muestra un ejemplo de función generadora para un conjunto de imágenes:

```
import os
import numpy as np
import cv2
def data_generator(batch_size, combinacion_arrays):
    categorias = os.listdir('./dataset128128')

    #print(categorias)
    url_fotos=[]
    #batch_size=32

    for x in categorias:
      if int(x) in combinacion_arrays:
          a=os.listdir('./dataset128128'+'/'+str(x))
          for url in a:
              url_fotos.append('./dataset128128'+'/'+str(x)+'/'+url)

    while True:
            np.random.shuffle(url_fotos)
            for start in range(0, len(url_fotos)-batch_size,
batch_size):

                train_X=[]
                train_Y=[]
                for x in range(start,start+batch_size):

                    imagen_cv2 = cv2.imread(url_fotos[x])
                    imagen_numpy = np.array(imagen_cv2)
                    train_X.append(imagen_numpy)
                    train_Y.append(int(url_fotos[x].split('/')[2]))
                train_Y = np.searchsorted(combinacion_arrays, train_Y)
                train_X = np.array(train_X)
                train_X=train_X / 255.0
                train_Y=np.array(train_Y)
                yield train_X, train_Y

#[...]
#Uso en el fit
steps_per_epoch = num_fotos//batch_size
#num_fotos es el número de fotos total o de las categorías de la subred
trained_model = model.fit(data_generator(batch_size,_DATA_TRAIN_GEN_Y ),
steps_per_epoch=steps_per_epoch, epochs=1 )
```

1.6. Preparación para uso con skynnet

El fichero de salida de la herramienta sk_tool se puede ejecutar de manera secuencial en una computadora, pero contiene código compatible con SkyNNet para su ejecución en paralelo y distribuido en una red de computadoras, a continuación se detallan los elementos necesarios para esto:

- Variables nonshared: Como se ejecuta en distintas máquinas, se usan variables globales en cada computador, estas son las variables nonshared, que son las variables de carga de datos, el modelo que se va a entrenar y usar, es una lista de tantos modelos como subredes habrá y la variable auxiliar to_predict_models que será una lista de cada modelo entrenado que tenga cada computador distribuido, para saber que modelos son los que tiene que usar cuando se lo pidan.

```
#__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]

#Noramalize the pixel values by deviding each pixel by 255
x_train = x_train / 255.0
x_test =  x_test / 255.0

model = [None, None, None]
to_predict_models = []
```

- Variables globales: Estas variables son únicas para todas las computadoras distribuidas, se genera una variable predictions_n donde n es el índice del bloque skynnet al que se refiere, empezando desde 0. En este ejemplo hay un bloque skynnet y una variable predictions_0 que almacenará todas las predicciones de las subredes para poder componerlas en una única predicción.

```
#__CLOUDBOOK:GLOBAL__
predictions_0 = {}
```

- Funciones paralelas (Etiqueta parallel): Las funciones de creación/entrenamiento y uso de las redes se hacen paralelas, se ejecutarán en las distintas computadoras a la vez.
- Funciones locales (Etiqueta local): Estas funciones se ejecutan de manera local, lo que significa que estarán en todas las computadoras distribuidas y no será necesario que haya comunicaciones entre máquinas con sus resultados.
- Etiqueta begin/end remove: Esta etiqueta incluye código que no se tiene en cuenta a la hora de usar SkyNNet, solo se usa cuando se ejecuta el fichero en modo local y se entrenan las redes de manera secuencial.

- Etiqueta lock/unlock: Esta etiqueta se usa para secciones críticas, que aunque estén trabajando en paralelo, pueden trabajar con una variable que no permita accesos simultáneos.
- Funciones DU0: Estas funciones se van a ubicar en el código del computador que inicie la ejecución del programa, al que se refiere como la du0: Deployable Unit 0 (Unidad desplegable 0)
- Etiqueta Sync: Esta etiqueta se usa después de invocar funciones paralelas, para esperar a que todas terminen. Como puede ser, no avanzar en el programa hasta que todas las subredes esten entrenadas.

```
for i in range(3):
      skynnet_train_0(i)#Esta función es paralela
#__CLOUDBOOK:SYNC__
```

- Etiqueta Main: Esta etiqueta establece el punto de entrada del programa.

1.7. Carga y guardado de modelos pre-entrenados

Si una aplicación carga un modelo pre entrenado, estará cargando el modelo original. Dicho modelo no se puede deconstruir a menos que sea un modelo creado por la misma aplicación en una ejecución anterior, en cuyo caso se podrían cargar los distintos submodelos en su lugar.

Por tanto la limitación de skynnet es la de trabajar con modelos pre entrenados no creados por la aplicación.

Para trabajar con submodelos pre entrenados se va a proceder de la siguiente manera:

- sk_tool va a traducir siempre la ruta de las invocaciones a load y save por una nueva ruta en la que el nombre del fichero incluya el índice de la subred.

```
#ORIGINAL
model = tf.keras.models.load_model('model.h5')
model.save('model.h5')
#PRODUCIDO POR SK_TOOL
model[sk_i] = tf.keras.models.load_model('model' + str(sk_i) + '.h5')
model.save('model' + str(sk_i) + '.h5')
```

1.8. Códigos de ejemplo

En la carpeta inputs se encuentran cinco códigos con pruebas de concepto (usadas para validar las estrategias del documento E.2.1), el código mostrado en este manual es de estos ficheros. Estos códigos son:

- sk_tool/input/mlp2.py : Del ejemplo a continuación
- sk_tool/input/bin_orig.py: Ejemplo de clasificación binaria
- sk_tool/input/autoenc_orig.py: Ejemplo problema regresión
- sk_tool/input/rnn_orig.py: Ejemplo red recurrente en problema clasificación
- sk_tool/input/cnn_orig.py: Ejemplo red convolucional en problema clasificación

- sk_tool/input/transformer_orig.py: Ejemplo uso transformers en problema clasificación

A continuación se muestra un ejemplo de un *mlp* con medidas de accuracy y loss

```
import os
import tensorflow as tf, numpy as np
import matplotlib.pyplot as plt
import time

#__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]

#Normalize the pixel values by dividing each pixel by 255
x_train = x_train / 255.0
x_test =  x_test / 255.0

#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
_NEURON_1 = 128
_NEURON_2 = 60
_NEURON_3 = 10
_EPOCHS = 10

#Modelo funcional
inputs = tf.keras.Input(shape=(28,28))
x = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(_NEURON_1, activation='relu')(x)
x = tf.keras.layers.Dense(_NEURON_2, activation='relu')(x)
outputs =  tf.keras.layers.Dense(_NEURON_3, activation='softmax')(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)

print(model.summary())
start=time.time()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(_DATA_TRAIN_X, _DATA_TRAIN_Y, validation_split=0.3,
epochs=_EPOCHS)
end=time.time()
```

```
print (" tiempo de training transcurrido (segundos) =", (end-start))

predicted = model.predict(_DATA_TEST_X)
#SKYNNET:END
```

1.9. 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
  - **_FILTER_**: 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
  - **_EPOCH_**: Variable que indica las épocas, no va en ninguna capa, es de la función fit. **IMPORTANTE: *En caso de deconstrucción de redes donde no se debe reducir las épocas (con capas convolucionales, clasificación binaria o regresión) no es conveniente usar esta variable, ya que automáticamente se reducirán. En su lugar se recomienda usar una constante o cualquier otro nombre de variable***

- 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.

1. Anexo: 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]

(*) si hay capas embedding, ni las capas densas ni las conv deben reducirse