Referencia del paquete dataLoader

Paquete que define las clases y funciones necesarias para el cargado de los datos.

Clases

class **genDataset**

Funciones

def **splitAndLoader** (dataset, val_split, batch_size, random_seed=42)

Función que genera una partición de validación y transforma el dataset en un 'DataLoader' de Pytorch.

Descripción detallada

Paquete que define las clases y funciones necesarias para el cargado de los datos.

Define la clase '**genDataset**', la cual hereda de la clase 'Dataset' de PyTorch, a fin de cargar en él los datos. La misma almacena los datos en memoria con formato NumPy y sólo transforma a Tensor de PyTorch los que está usando en el momento para pasar la red. Luego, define la función '**splitAndLoader()**' que toma el dataset generado para crear un 'DataLoader' de PyTorch, el cual provee facilidades para iterar por batch sobre el dataset. Además, la función anterior reserva una partición de los datos para validación, si así se indicase.

Documentación de las funciones

def dataLoader.splitAndLoader (*dataset*, *val_split*, *batch_size*, *random_seed* = `42`)

Función que genera una partición de validación y transforma el dataset en un 'DataLoader' de Pytorch.

Si el porcentaje de los datos a reservar para validación 'val_split' es mayor a 0, se generan dos DataLoaders. Uno para validación con el porcentaje indicado y otro para entrenar con lo restante. Esta división se realiza mezclando los índices de los datos con la función 'shuffle()' de NumPy, a la cual se le puede proporcionar una semilla 'random_seed' para poder hacer reproducible el experimento. Luego, con la función 'SubsetRandomSampler()' de PyTorch se generan las dos particiones para crear los DataLoaders. A éstos últimos también debe pasársele el tamaño de batch que se utilizará.

Parámetros:

dataset	Dataset de la clase definida anteriormente al cual se procesará.
val_split	Porcentaje de datos reservados para validación. Si fuese 0, no se reserva ninguno y se devuelve sólo un DataLoader con todos los datos.
batch_size	Tamaño de batch que usará el DataLoader.
random_seed	Semilla que se le pasa a NumPy para hacer reproducible el experimento. Por defecto, es igual a 42.

Devuelve:

Una tupla que contiene el DataLoader de entrenamiento y el DataLoader de validación, o simplemente un único DataLoader con todos los datos si 'val_split' es 0.

Referencia del archivo inferir

Script que itera sobre los datos pasados como parámetro para calcular la salida de la red convolucional.

Descripción detallada

Script que itera sobre los datos pasados como parámetro para calcular la salida de la red convolucional.

Es una versión simplificada del archivo **'trainModel.py'** ya que simplemente infiere sobre los datos, no realiza partición de validación ni utiliza datos de test o de cromosomas reales. Además, permite la posibilidad de realizar la inferencia sin conocer el ground truth de la imagen mediante el parámetro 'hayLabels'. Otra diferencia es que este archivo obviamente no realiza gráficas de pérdida, recall y Jaccard por época, sino que sólo las calcula en el caso de que haya ground truth.

Tampoco carga el optimizador ya que la idea no es continuar un entrenamiento.

Referencia del archivo inferir_params.py

Script que define los parámetros que se utilizan para realizar inferencias con la red convolucional.

Descripción detallada

Script que define los parámetros que se utilizan para realizar inferencias con la red convolucional.

En la siguiente tabla se detallan los parámetros, indicando los que trae por defecto (que fueron utilizados para inferir con la red W sobre cromosomas reales).

Parámetro	Detalle	Valor por defecto
batch_size	Tamaño del batch en que se toman los datos para pasar por la red convolucional.	10
nombreArq	Nombre de la arquitectura de red (sin '.py').	"RedW"
nombreData	Nombre de los datos (sin '.npz')	"reales"
hayLabels	Booleano que indica si los datos poseen ground truth o no.	False
cant_train	Cantidad de archivos de entrenamiento usados (sólo cuando se usan archivos de entrenamiento)	1
nameData	Cadena de texto que se le agrega a los nombres de los archivos de salida	""
pathAnterior	Directorio de los parámetros entrenados del modelo, utilizado para continuar un entrenamiento previo.	"./modelo/entrenado"
DATA_PATH	Carpeta en que se encuentran los datos	"./data/"
OUT_PATH	Directorio en que se guardan los archivos de salida	"./salida/"+nameData
PREDICT_PATH	Directorio en que se guardan las predicciones de la red como imágenes	"./salida/"+"predicciones/" + nameData
verGraficas	Booleano que indica si se quieren ver las gráficas online	False
cadaCuantas	Entero que indica cada cuantas imágenes guardar las máscaras predichas.	1

Referencia del paquete modelLoader

Paquete que define las funciones necesarias para el cargado y guardado de un modelo entrenado.

Funciones

def **saveModel** (epoch, model, optimizer, is_best=False, path="./model/")

Función que guarda el modelo en el directorio indicado con la función 'save()' de PyTorch.

def **loadModel** (path, model, optimizer=None)

Función que carga el modelo desde el directorio indicado con la función 'load()' de PyTorch.

def **getOptimizer** (model, name, learning_rate=0.001, momentum=0.9)

Función que devuelve el optimizador de PyTorch requerido.

def **getLossFunc** (name)

Función que devuelve la función de pérdida de PyTorch requerida.

Descripción detallada

Paquete que define las funciones necesarias para el cargado y guardado de un modelo entrenado.

Además, define funciones para obtener funciones de pérdida y optimizadores.

Documentación de las funciones

def modelLoader.getLossFunc (*name*)

Función que devuelve la función de pérdida de PyTorch requerida.

Pueden ser "BCE" (Binary Cross Entropy), "BCEwLogits" (Binary Cross Entropy with Logits), "CrossEntropy" y "MSE" (Mean Squared Error).

Parámetros:

name	Cadena de texto que indica la función a devolver.

Devuelve:

Función de pérdida indicada de PyTorch.

def modelLoader.getOptimizer (*model*, *name*, *learning_rate* = `0.001`, *momentum* = `0.9`)

Función que devuelve el optimizador de PyTorch requerido.

Pueden ser gradiente descendiente estocástico "SGD", "Adam" o "Adadelta". En el primer caso se debe pasar la tasa de aprendizaje 'learning_rate' y el momentum 'momentum'. En el segundo, sólo la tasa de aprendizaje. Mientras que en el último no necesita ninguno de los dos ya que son parámetros que aprende durante el entrenamiento.

Parámetros:

model	Arquitectura de PyTorch que se optimizará.
name	Cadena de texto que indica el optimizador que se quiere.
learning_rate	Tasa de aprendizaje.
momentum	Momentum, sólo utilizado en "SGD".

Devuelve:

Optimizador de PyTorch requerido.

def modelLoader.loadModel (*path*, *model*, *optimizer* = `None`)

Función que carga el modelo desde el directorio indicado con la función 'load()' de PyTorch.

Con la función 'load_state_dict()' del modelo de PyTorch se re-establecen los parámetros entrenados del modelo. Si se quiere cargar el optimizador para continuar el entrenamiento, se hace con la función 'load_state_dict()' del optimizador de PyTorch.

Parámetros:

path	Directorio del modelo a cargar.
model	Arquitectura de PyTorch del modelo a cargar.
optimizer	Optimizador del cual se quiere reestablecer el estado previo al guardado. Si es 'None', no se carga.

Devuelve:

Devuelve una tupla que contiene el número de época en que se cargó, el modelo con los parámetros cargados y el optimizador con el estado re-establecido.

def modelLoader.saveModel (*epoch*, *model*, *optimizer*, *is_best* = `False`, *path* = `"./model/"`)

Función que guarda el modelo en el directorio indicado con la función 'save()' de PyTorch.

Los parámetros entrenados se guardan en un diccionario con la clave 'state_dict' y el estado del optimizador como 'optim_dict', a fin de poder reanudar el entrenamiento. Además, almacena la época en que se hace con la clave 'epoch'.

Parámetros:

epoch	Entero que indica la época en que se guarda el modelo.
model	Modelo de PyTorch al cual se quieren almacenar los parámetros entrenados.
optimizer	Optimizador que se utilizó en el entrenamiento y se quiere almacenar el estado.
is_best	Booleano que indica si se está guardando el modelo con menor pérdida o no, para guardarlos diferenciadamente. En el primer caso, le añade "_best" al nombre y en el segundo "_final".
path	Directorio en que se almacena.

Referencia del archivo parameters.py

Script que define los parámetros que se utilizan para el entrenamiento de la red convolucional.

Descripción detallada

Script que define los parámetros que se utilizan para el entrenamiento de la red convolucional.

En la siguiente tabla se detallan los parámetros, indicando los que trae por defecto que fueron utilizados para entrenar la red W.

Parámetro	Detalle	Valor por defecto
num_epochs	Cantidad de épocas que se entrena.	30
batch_size	Cantidad de datos para pasar por la red simultáneamente.	10
learning_rate	Tasa de aprendizaje.	0,001
momentum	Momentum (sólo aplicado para función de pérdida SGD).	0,9
nombreArq	Nombre de la arquitectura de red (sin '.py').	"RedW"
tipoOptim	Optimizador usado para el entrenamiento.	"Adam"
tipoLoss	Función de pérdida usada para el entrenamiento.	"CrossEntropy"
cant_train	Cantidad de archivos de entrenamiento usados.	12
val_split	Porcentaje de datos reservados para la validación.	0,15
nameData	Texto que se agrega al nombre de los archivos de salida	""
pathAnterior	Directorio de los parámetros entrenados del modelo, (útil para reanudar un entrenamiento). Si es vacío, no se carga.	""
nameModel	Similar a 'nameData' para los parámetros entrenados.	""
DATA_PATH	Carpeta en que se encuentran los datos	"./data/"
MODEL_STORE_PATH	Directorio en que se guardan los parámetros del modelo entrenado.	"./modelo/"+tipoOptim+ "_"+tipoLoss + "_"+ nameModel
OUT_PATH	Directorio en que se guardan los archivos de salida.	"./salida/"+nameData
PREDICT_PATH	Directorio en que se guardan las predicciones como imágenes.	"./salida/"+"predicciones/" + nameData
REALES_PATH	Path para las predicciones de la red sobre datos reales.	"./inferencias/"+ nameData
verGraficas	Booleano que indica si se quieren ver las gráficas online.	False
cadaCuantas	Entero que indica cada cuantas épocas guardar las gráficas.	1
cadaCuantos	Indica cada cuantos batches guardar las predicciones.	300
cadaCuantosVal	Similar al anterior para validación.	60
cadaCuantosTest	Similar al anterior para test.	20

Referencia del paquete RedW

Paquete que incluye la clase '**RedW**' que define la arquitectura de la red W, similar a dos red U conectadas.

Clases

class **RedW**

Descripción detallada

Paquete que incluye la clase '**RedW**' que define la arquitectura de la red W, similar a dos red U conectadas.

Dicha red U se determina en 'OverlapSegmentationNet.py'. Se hereda de la clase 'nn.Module' de PyTorch y se sobrecarga la función que se encarga de hacer el pasaje hacia adelante de la imagen '**forward()**'.

Referencia del paquete resultados

Archivo en el que se definen las clases y funciones utilizadas para guardar los resultados parciales y finales que se obtienen.

Clases

class **graficas**

Funciones

def **inicializarLogger** (filename='log.txt')

Función que inicializa el logger utilizado para guardar cada paso en el entrenamiento.

def **log** (cadena, porConsola=True)

Función que guarda en el archivo de log la cadena que se le pasa.

def **medidas** (predicted, labels)

Función que calcula el recall y el Jaccard promedio de un batch de imágenes dados sus correspondientes ground truths.

Descripción detallada

Archivo en el que se definen las clases y funciones utilizadas para guardar los resultados parciales y finales que se obtienen.

Incluye el cálculo de las medidas de recall y Jaccard, así como también la graficación de ellos y la pérdida. Estas gráficas se realizan para el entrenamiento y para la validación mediante la librería 'matplotlib'. También incluye la muestra online de predicciones y el almacenamiento de ellas.

Documentación de las funciones

def resultados.inicializarLogger (*filename* = `'log.txt'`)

Función que inicializa el logger utilizado para guardar cada paso en el entrenamiento.

Utiliza la librería 'logging' que trae por defecto Python.

Parámetros:

filename	Nombre del archivo en el que se guarda.

def resultados.log (*cadena*, *porConsola* = `True`)

Función que guarda en el archivo de log la cadena que se le pasa.

Parámetros:

cadena	Cadena de texto que se guarda en el archivo de log.
porConsola	Booleano que indica si la cadena también se muestra por consola.

def resultados.medidas (*predicted*, *labels*)

Función que calcula el recall y el Jaccard promedio de un batch de imágenes dados sus correspondientes ground truths.

Para ello, primero calcula la clase de cada píxel mediante la función 'argmax()' de NumPy. A continuación, calcula la matriz de confusión mediante la función 'confusion_matrix()'. Luego, recorre cada fila (correspondiente a cada clase) y, si hay algo en ella, hace el cálculo de las medidas y las añade a una lista. Si son todos ceros significa que esa clase existe pero nunca fue acertada y traería problemas de división por cero. En ese caso, directamente se añade un recall y Jaccard de valor 0. Se calcula el recall como el valor de la diagonal dividido la suma de la fila [TP / (TP+FN)]. El Jaccard es el valor de la diagonal dividido la suma de la fila y de la columna

[TP / (TP+FN+FP)]. Entre corchetes se indica la fórmula, significando TP verdaderos positivos, FN falsos negativos y FP falsos positivos. Por último, se promedia el recall y el Jaccard y se devuelve como tupla.

Parámetros:

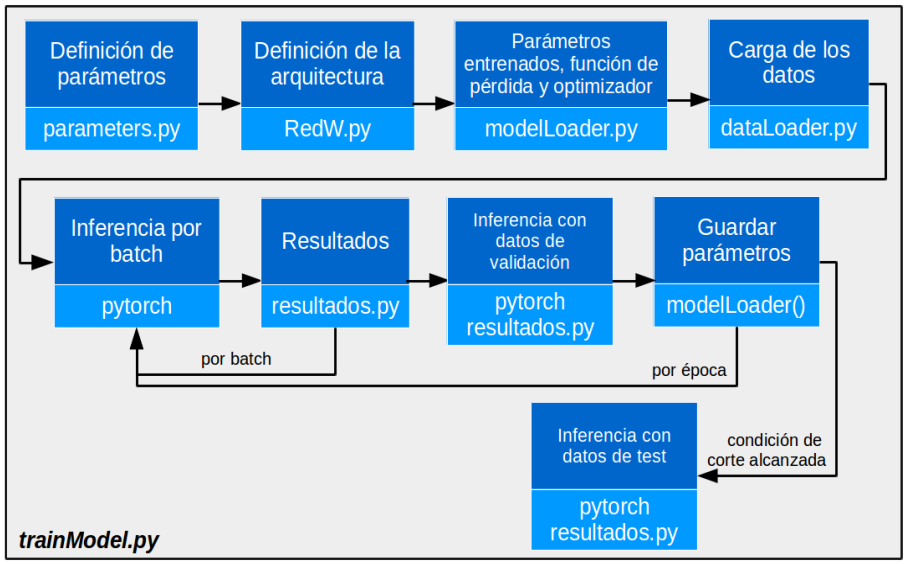
predicted	Arreglo 4D de NumPy. La primera corresponde a la cantidad de datos, la segunda a la cantidad de canales (clases) y las restantes al tamaño de la imagen.
labels	Arreglo de NumPy 3D que contiene el número de clase de cada píxel. La primera dimensión es la cantidad de datos y las restantes el tamaño de la imagen. El número indica la clase correcta de cada clase o equivalentemente el canal de 'predicted' correcto.

Devuelve:

Tupla conteniendo los valores promedio de recall y Jaccard.

Referencia del archivo trainModel

Script que itera sobre los datos de entrenamiento para entrenar la red convolucional. En la siguiente imagen se ve un diagrama de bloques de cómo funciona el mismo.



Los parámetros se cargan desde el archivo ‘parameters.py’ en que se definen. La arquitectura de la red convolucional utilizada se determina en ‘RedW.py’ mediante la librería PyTorch. Luego, se utilizan las funciones definidas en ‘modelLoader.py’ para obtener la función de pérdida ‘getLossFunc()’, obtener el optimizador ‘getOptimizer()’ y para cargar parámetros ya entrenados para el modelo (si los hubiera) ‘loadModel()’. También se inicializa el logger en el que se guardará datos referidos a cada batch y la clase graficas definida en ‘resultados.py’.

Se cargan los datos de train mediante la clase ‘genDataset()’ definida en ‘dataLoader.py’. Luego, con la función ‘splitAndLoader()’ también definida en ‘dataLoader.py’, se divide el dataset en una partición de entrenamiento y otra de validación. Ambas se transforman ahí mismo en un DataLoader definido por PyTorch, que permite iterar fácilmente por batch sobre cada partición. Un procedimiento similar se realiza con las imágenes de cromosomas reales (sin particionar).

A continuación, se define un bucle en el que se recorren los datos de train de a batches, se envían a la GPU si estuviera disponible mediante ‘to()’, se infiere sobre las imágenes del mismo con la función ‘forward()’ mencionada anteriormente, se actualizan los parámetros según la pérdida calculada con la función ‘backward()’ y se guardan los resultados obtenidos mediante la clase graficas definida en ‘resultados.py’. Esta última posee una función ‘cambiarPredicted()’ utilizada para guardar predicciones que realiza la red. También se acumulan en listas las medidas de recall y coeficiente de Jaccard obtenidos mediante la función ‘medidas()’ definida en ‘resultados.py’. Luego, se acumula en otras listas las medidas anteriores promediadas por época y mediante la función ‘graficar()’ de ‘graficas()’ se genera una figura que muestra gráficas del progreso de las medidas mencionadas en entrenamiento y validación durante las épocas. Sumado a lo anterior, con la función ‘log()’ definida en ‘resultados.py’ se actualiza en OUT PATH un archivo ’log.txt’ con las medidas en cada batch. Antes de proseguir con la siguiente iteración, se guardan los parámetros entrenados si es que la pérdida promedio en validación es menor a la mejor pérdida promedio obtenida hasta el momento. Esto se hace mediante la función ‘saveModel()’ de ‘modelLoader.py’.

Luego, se repite el proceso desde el recorrido de los datos de entrenamiento hasta alcanzar la cantidad de épocas límite definida en ‘parameters.py’, que es la única condición de corte del entrenamiento. Una vez finalizado, se realiza un procedimiento similar con las imágenes de test.

Documentación de las clases

Referencia de la Clase dataLoader.genDataset

Métodos públicos

def **__init__** (self, paraQue, hayLabels=True, cant_train=1, path="./data/")

Constructor de la clase que lee los datos del disco.

def **__getitem__** (self, index)

Sobrecarga del método homólogo de la clase 'Dataset' de Pytorch.

def **__len__** (self)

Sobrecarga del método homólogo de la clase 'Dataset' de Pytorch.

Documentación del constructor

def dataLoader.genDataset.__init__ (*self*, *paraQue*, *hayLabels* = `True`, *cant_train* = `1`, *path* = `"./data/"`)

Constructor de la clase que lee los datos del disco.

Si los datos fueran para entrenar (para "train") carga en 'data' todos los archivos con forma "trainN.npz", donde N es un número del 1 a 'cant_train'. De lo contrario, simplemente carga el archivo indicado. También se le pasa como parámetro si los datos tienen ground truth o no, para que los cargue o no en 'labels'. Además, define la transformación que se le realizarán a los datos para normalizarlos con media 0.5 y desvío 0.5 mediante la librería 'torchvision'.

Parámetros:

paraQue	String que indica si es para "train" o en otro caso es el nombre del archivo (sin '.npz').
hayLabels	Booleano que indica si los datos tienen ground truth o no.
cant_train	Entero que indica la cantidad de archivos de entrenamiento en el caso de "train".
path	Carpeta en la que se encuentran los mencionados archivos.

Documentación de las funciones miembro

def dataLoader.genDataset.__getitem__ (*self*, *index*)

Sobrecarga del método homólogo de la clase 'Dataset' de Pytorch.

Define cómo se obtiene una imagen y su ground truth (si hubiera) del dataset. Además, los transforma a Tensor de PyTorch y aplica la transformación definida en '**__init__** ()'.

Parámetros:

index	Índice del dato a obtener.

Devuelve:

Imagen y ground truth (si hubiera) como Tensor de PyTorch de 3 y 4 dimensiones.

def dataLoader.genDataset.__len__ (*self*)

Define cómo se obtiene la longitud del dataset.

La documentación para esta clase fue generada a partir del siguiente fichero:

0 dataLoader.py

Referencia de la Clase resultados.graficas

Métodos públicos

def **__init__** (self, online=True, size=(15, 8))

Constructor de la clase que inicializa la figura con las subfiguras en las que se graficará.

def **cambiarPredicted** (self, epoch, maskPredicted1, maskPredicted2, path="./", name="", real=False)

Actualiza las predicciones en la figura si 'online' es 'True', sino sólo las almacena como archivo.

def **graficar** (self, epoch, loss, acc, jacc, loss_val, acc_val, jacc_val, path="./", guardar=True, ultimo=False, name="")

Función que actualiza las gráficas de pérdida, recall y Jaccard de entrenamiento y de validación.

Documentación del constructor y destructor

def resultados.graficas.__init__ (*self*, *online* = `True`, *size* = `(15,8)`)

Constructor de la clase que inicializa la figura con las subfiguras en que se graficará.

Si 'online' es falso, las subfiguras son 6. De izquierda a derecha corresponden a la pérdida, recall y Jaccard promedio por época. Las tres superiores son de entrenamiento y las tres inferiores de validación. Si 'online' es verdadero, las subfiguras son 9 ya que se muestran también predicciones parciales de la red.

Parámetros:

online	Booleano que indica si las gráficas se muestran mientras se entrena o solamente se guardan en un archivo por época.
size	Tamaño de la figura.

Documentación de las funciones miembro

def resultados.graficas.cambiarPredicted (*self*, *epoch*, *maskPredicted1*, *maskPredicted2*, *path* = `"./"`, *name* = `""`, *real* = `False`)

Actualiza las predicciones en la figura si 'online' es 'True', sino sólo las almacena como archivo.

Parámetros:

epoch	Número de época utilizado para guardar las predicciones.
maskPredicted1	Arreglo 2D de NumPy correspondiente a la imagen que se quiere mostrar y guardar. Generalmente es la predicción de la red.
maskPredicted2	Arreglo 2D de NumPy correspondiente a la imagen que se quiere mostrar y guardar. Generalmente es el ground truth o la imagen de la que se predijo.
path	Directorio en que se guardan 'maskPredicted1' y 'maskPredicted2'.
name	Cadena de texto extra que se agrega al nombre de la magen al guardarla.
real	Booleano que si es verdadero guarda 'maskPredicted2' en escala de grises por corresponder a cromosomas, sino se guarda en colores por corresponder a una máscara que indica las clases.

def resultados.graficas.graficar (*self*, *epoch*, *loss*, *acc*, *jacc*, *loss_val*, *acc_val*, *jacc_val*, *path* = `"./"`, *guardar* = `True`, *ultimo* = `False`, *name* = `""`)

Función que actualiza las gráficas de pérdida, recall y Jaccard de entrenamiento y de validación.

Además, si se indica, se guardan las medidas obtenidas en un archivo '.npz' con claves iguales a los nombres de los parámetros.

Parámetros:

epoch	Número de época utilizado para guardar las predicciones.
loss	Lista que contiene la pérdida de entrenamiento en cada época.

acc	Lista que contiene el recall de entrenamiento en cada época.
jacc	Lista que contiene el Jaccard de entrenamiento en cada época.
loss_val	Lista que contiene la pérdida de validación en cada época.
acc_val	Lista que contiene el recall de validación en cada época.
jacc_val	Lista que contiene el Jaccard de validación en cada época.
guardar	Booleano que indica si se guardan las medidas en un archivo '.npz' o no.
ultimo	Booleano que indica si es la última gráfica, entonces espera que se aprete un botón para cerrarse.
name	Cadena de texto extra que se agrega al nombre de la magen al guardarla.

La documentación para esta clase fue generada a partir del siguiente fichero:

1 resultados.py

Referencia de la Clase OverlapSegmentationNet.OverlapSegmentationNet

Métodos públicos

def **__init__** (self, canalesEntrada=1)

Constructor que define la estructura de la red convolucional.

def **forward** (self, x)

Sobrecarga de la función homóloga de PyTorch que realiza la pasada hacia adelante de la red.

Atributos públicos

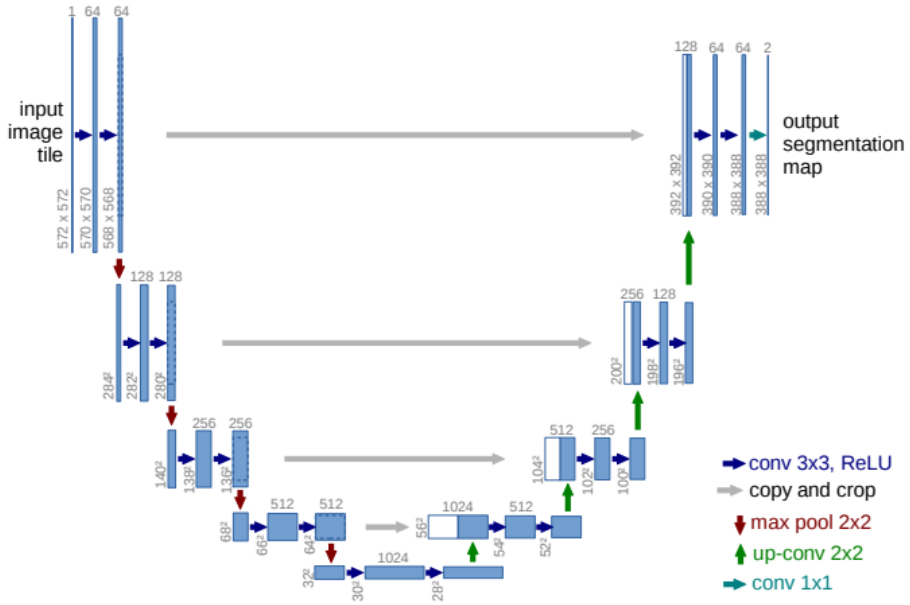
canalesEntrada: determina cuantos canales de entrada se definirán en la primera capa del modelo

---

Documentación del constructor y destructor

def OverlapSegmentationNet.OverlapSegmentationNet.__init__ (*self*, *canalesEntrada* = 1)

Constructor que define la estructura de la red convolucional. Ésta es similar a la red U de la siguiente imagen.



Documentación de las funciones miembro

def OverlapSegmentationNet.OverlapSegmentationNet.forward (*self*, *x*)

Sobrecarga de la función homóloga de PyTorch que realiza la pasada hacia adelante de la red.

Se encarga de determinar cómo se relacionan entre sí las capas definidas en el constructor.

Parámetros:

x	Entradas en formato Tensor de 4 dimensiones de PyTorch. La primera corresponde a la cantidad de datos, la segunda a la cantidad de canales y las últimas dos al tamaño de la imagen.

Devuelve:

Salida de la red convolucional en formato Tensor 4D de PyTorch. La primera corresponde a la cantidad de datos, la segunda a la cantidad de canales (24 clases) y las últimas dos al tamaño de la imagen.

La documentación para esta clase fue generada a partir del siguiente fichero:

2 OverlapSegmentationNet.py

Referencia de la Clase RedW.RedW

Métodos públicos

def **__init__** (self)

Constructor que define la estructura de la red convolucional.

Documentación del constructor y destructor

def RedW.RedW.__init__ (*self*)

Constructor que define la estructura de la red convolucional. La misma consta de dos redes U definidas en 'OverlapSegmentationNet.py', en la que se conecta la salida de una a la entrada de la otra.

Documentación de las funciones miembro

def RedW.RedW.forward (*self*, *x*)

Sobrecarga de la función homóloga de PyTorch que realiza la pasada hacia adelante por la red convolucional.

Se encarga de determinar cómo se interrelacionan entre sí las capas definidas anteriormente en el constructor.

Parámetros:

x	Entradas en formato Tensor de 4 dimensiones de PyTorch. La primera corresponde a la cantidad de datos, la segunda a la cantidad de canales y las últimas dos al tamaño de la imagen.

Devuelve:

Salida de la red convolucional en formato Tensor de 4 dimensiones de PyTorch. La primera corresponde a la cantidad de datos, la segunda a la cantidad de canales (24 por la cantidad de clases) y las últimas dos al tamaño de la imagen.

La documentación para esta clase fue generada a partir del siguiente fichero:

3 RedW.py